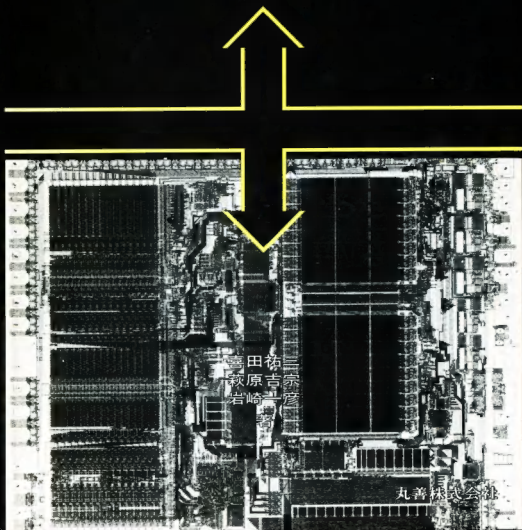


森亮一 監修  
マイクロコンピュータシリーズ

14

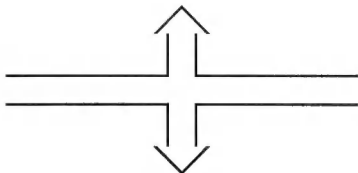
# 68000 マイクロコンピュータ





森亮一 監修  
マイクロコンピュータシリーズ **14**

# 68000 マイクロコンピュータ



喜田祐三  
萩原吉宗  
岩崎一彦  
著

丸善株式会社



## 序

1971年マイクロコンピュータが誕生してから、すでに10年が過ぎた。そして、今日、マイクロコンピュータは広範な分野において多様な形で私達の社会生活を支えるようになったといっても過言ではない。今や、私達の生活の中には無意識のうちにマイクロコンピュータの恩恵を受けている部分がいかに多いか知るべきである。さて、マイクロコンピュータはこの10年間、MOS-LSI技術の進歩に支えられて、集積度、スピードはともに30~50倍改善され、また消費電力も1/10以下に低減化された。前者は主としてLSIの微細化技術の進歩に負うところが大きく、一方後者はデバイス、回路技術の進歩に負うところが大きい。マイクロコンピュータは大別すると(1)多機能シングルチップ、(2)高性能マルチチップ、の2つに分類される。前者は主として家電・民生分野に多く使用され、私達の生活の中に深く浸透している。たとえば、テレビジョンはマイクロコンピュータの力によって美しい画面に自動的に最適同調され、ルームエアコンは温度や湿度を検知して快適な環境を最もエネルギーを節約した形で実現する。一方、後者は高機能・高性能ゆえに多くは産業分野において使用されている。制御、計測、通信、データ処理、工業などでの実績を踏まえて、最近の1つの特徴はOA、パーソナルコンピュータへの展開にあり、徐々に私達の生活に密着した分野に広まってきているといえる。特にパーソナルコンピュータは人間との対話を中心であるため、マイクロコンピュータのトータルパフォーマンスや価格の他にハード、ソフト両面の使いやすさが重要課題となる。また、パーソナルコンピュータの普及による実績と技術の蓄積は次の時代にくるであろうホームコンピュータ普及への基盤を創生するものと思われる。

従来、パーソナルコンピュータは高性能8ビットマイクロコンピュータが使用されてきたが、最近では8ビットから16ビットマイクロコンピュータにその対象機種が移行しつつあり、ここ1~2年のうちに16ビットマイクロコンピュータに多くが移行するものと予

想される。

さて、現在 16 ビット マイクロコンピュータは 68000, 8086, Z 8000 など数機種が世界の主流を占めているが、なかでも 68000 は現在入手し得る 16 ビット マイクロコンピュータの最も高性能な機種であり、その能力の高さと使いやすさにより、今後広く普及するものと思われる。

本書は 16 ビット マイクロコンピュータ 68000 のハードウェアからソフトウェア、さらにシステム開発サポートまでを含めて解説したものである。68000 を使用する技術者が、その内容と使い方を容易に理解し、実践できるように工夫した。本書は大きく分けてⅢ編から成る。第Ⅰ編（1 章～10 章）はハードウェアの解説である。ハードウェアはバスインターフェイス、入出力インターフェイス、例外処理、DMA 転送技術などについて特に注意を払った。第Ⅱ編（11 章～19 章）はソフトウェアの解説である。ソフトウェアはできるかぎり多くの例題を取り入れることにより理解を深めることに留意した。第Ⅲ編（20 章～22 章）は開発サポートに関する解説である。全編を通じて、68000 を使用する技術者、学生の立場に立って、例題を利用しつつ疑問点に対する解釈という視点で執筆を行った。本書が 16 ビット マイクロコンピュータ 68000 使用技術者の座右の書として役立つことを願っている。

最後に、本書刊行に当って御指導いただいた筑波大学 森亮一教授、日立製作所半導体事業部 初鹿野晃一氏、同 武蔵工場 中野浩行氏、石川知雄氏、ならびに丸善株式会社出版部各位に対し深謝の意を表する。

1983 年 2 月

著 者

# 執 筆 者

喜 田 祐 三	株式会社 日立製作所
萩 原 吉 宗	〃
岩 崎 一 彦	〃
中 村 英 夫	〃
塚 田 敏 郎	〃
船 橋 恒 男	〃
山 口 昇	〃
高 木 克 明	〃
沢 瀬 照 美	〃
松 浦 達 治	〃
野 口 孝 樹	〃
志 村 隆 則	〃





# 目 次

## I 68000 のハードウェア

1 ま え が き .....	2
1.1 マイクロコンピュータの進歩 .....	2
1.2 68000 の 概 要 .....	4
2 68000 のアーキテクチャ .....	6
2.1 アーキテクチャの特徴 .....	6
2.2 プロセッサの内部構造 .....	7
2.3 レジスタの構成 .....	9
2.3.1 データレジスタ (D0~D7) .....	10
2.3.2 アドレスレジスタ (A0~A6), システムスタックポインタ (A7) .....	10
2.3.3 プログラムカウンタ (PC) .....	12
2.3.4 ステータスレジスタ .....	13
2.4 データの構成 .....	14
2.4.1 メモリ内のデータの構成 .....	14
2.4.2 整数データの表現形式 .....	15
2.5 基本システム構成と動作 .....	17
2.5.1 入出力インターフェイスの構成と特徴 .....	17
2.5.2 基本動作サイクル .....	19
2.5.3 命令ブリフエッチ機能 .....	19
2.5.4 基本命令シーケンス .....	20

3 インターフェイス信号とバスオペレーション .....	22
3.1 インターフェイス信号 .....	22
3.1.1 アドレスバス, データバス .....	22
3.1.2 非同期バス制御信号 .....	23
3.1.3 バスアービトラレーション制御信号 .....	25
3.1.4 割込み制御信号 .....	25
3.1.5 システム制御信号 .....	25
3.1.6 6800 周辺 LSI 制御信号 .....	26
3.1.7 プロセッサステータス .....	28
3.1.8 クロック (CLK) .....	28
3.1.9 信号のまとめ .....	28
3.2 バスオペレーション .....	28
3.2.1 データ転送オペレーション .....	28
3.2.2 バスアービトラレーション .....	35
3.2.3 バスエラーおよびホールド オペレーション .....	38
3.2.4 リセット オペレーション .....	41
4 命令の形式とアドレス形式 .....	43
4.1 命令の形式 .....	43
4.2 アドレス形式 .....	44
4.2.1 データレジスタ直接形式 .....	45
4.2.2 アドレスレジスタ直接形式 .....	45
4.2.3 アドレスレジスタ間接形式 .....	46
4.2.4 ポストインクリメントアドレスレジスタ間接形式 .....	46
4.2.5 プリデクリメントアドレスレジスタ間接形式 .....	47
4.2.6 ディスプレースメント付アドレスレジスタ間接形式 .....	48
4.2.7 インデックス付アドレスレジスタ間接形式 .....	49
4.2.8 短絶対アドレス形式 .....	50
4.2.9 長絶対アドレス形式 .....	51
4.2.10 ディスプレースメント付プログラムカウンタ相対形式 .....	52

4.2.11 インデックス付プログラム カウンタ相対形式 .....	52
4.2.12 イミディエイト形式 .....	53
4.2.13 クイック イミディエイト形式 .....	55
4.2.14 インブライド形式 .....	55
<b>5 命令の種類</b> .....	<b>57</b>
5.1 データ転送命令 .....	57
5.2 算術演算命令 .....	60
5.3 論理演算命令 .....	62
5.4 2進化10進数演算命令 .....	62
5.5 桁移動操作命令 .....	63
5.6 ビット操作命令 .....	64
5.7 プログラム制御命令 .....	65
5.8 システム制御命令 .....	67
<b>6 フラグと算術演算</b> .....	<b>70</b>
6.1 命令実行とフラグの設定 .....	70
6.2 各フラグの説明 .....	71
6.2.1 N フ ラ グ .....	71
6.2.2 Z フ ラ グ .....	73
6.2.3 V フ ラ グ .....	74
6.2.4 C フ ラ グ .....	76
6.2.5 X フ ラ グ .....	77
<b>7 命令実行時間</b> .....	<b>78</b>
7.1 命令実行時間の内訳 .....	78
7.2 実効アドレス計算時間 .....	79
7.3 オペレーション実行処理時間 .....	80
7.3.1 データ転送命令実行時間（実効アドレス計算処理時間を含む） .....	80
7.3.2 標準命令実行時間 .....	82
7.3.3 イミディエイト命令実行時間 .....	83

7.3.4 シングルオペランド命令実行時間 .....	84
7.3.5 シフト/ローテート命令実行時間 .....	84
7.3.6 ビット操作命令実行時間 .....	86
7.3.7 ブランチ関係命令実行時間 .....	86
7.3.8 ジャンプ関係命令実行時間 .....	88
7.3.9 実効アドレス生成関係命令実行時間 .....	88
7.3.10 マルチレジスタ関係命令実行時間 .....	88
7.3.11 倍精度演算命令実行時間 .....	90
7.3.12 その他の命令実行時間 .....	90
7.3.13 例外処理時間 .....	91
8 例外処理 .....	92
8.1 プロセッサ処理状態 .....	92
8.2 プログラム実行状態 .....	93
8.3 例外処理 .....	95
9 周辺デバイスとのインターフェイス .....	110
9.1 最小システム構成 .....	111
9.2 ROM およびスタティック RAM とのインターフェイス .....	113
9.3 ダイナミック RAM とのインターフェイス .....	114
9.4 割込みベクタ発生回路 .....	115
9.5 6800 周辺 LSI とのインターフェイス .....	116
10 68000 ファミリー周辺 LSI .....	120
10.1 68450 DMAC (Direct Memory Access Controller) .....	120
10.1.1 68450 DMAC の特徴 .....	120
10.1.2 信号線とバスサイクル .....	121
10.1.3 転送プロトコル .....	124
10.1.4 68450 DMAC を使用したシステム構成例 .....	126
10.2 68451 MMU (Memory Management Unit) .....	127
10.2.1 68451 MMU の特徴 .....	127

10.2.2 信号線とアドレス変換のバスサイクル .....	128
10.2.3 アドレス変換 .....	130
10.3 68120 IPC (Intelligent Peripheral Controller) .....	132
10.4 68230 PI/T (Parallel Interface/Timer) .....	135

## II 68000 のソフトウェア

11 アセンブラ .....	138
11.1 ソースプログラムの構造 .....	139
11.2 アセンブラの書式 .....	139
11.2.1 ラベルフィールド .....	139
11.2.2 演算フィールド .....	140
11.2.3 オペランドフィールド .....	141
11.2.4 コメントフィールド .....	141
11.3 データ形式 .....	141
11.3.1 数値定数 .....	141
11.3.2 文字定数 .....	142
11.3.3 記号 .....	142
11.3.4 式 .....	143
11.4 実効アドレスとアドレス形式 .....	144
11.4.1 データレジスタ直接形式 .....	145
11.4.2 アドレスレジスタ直接形式 .....	145
11.4.3 アドレスレジスタ間接形式 .....	145
11.4.4 ポストインクリメントアドレスレジスタ間接形式 .....	145
11.4.5 プリデクリメントアドレスレジスタ間接形式 .....	146
11.4.6 ディスプレースメント付アドレスレジスタ間接形式 .....	146
11.4.7 インデックス付アドレスレジスタ間接形式 .....	146
11.4.8 絶対アドレス形式 .....	147
11.4.9 ディスプレースメント付プログラムカウンタ相対形式 .....	147
11.4.10 インデックス付プログラムカウンタ相対形式 .....	147
11.4.11 イミディエイトデータ形式 .....	148
11.4.12 CCR/SR 形式 .....	148

11.5	アセンブラ制御命令	148
11.6	リロケートブルなプログラミング	150
12	データ転送命令	152
12.1	データ転送命令	152
12.2	データ転送命令の応用例	161
12.2.1	単純なデータ転送	161
12.2.2	ブロック転送	161
12.2.3	プログラムスタック	162
12.2.4	キュー (QUEUE)	164
13	算術論理演算命令	166
13.1	算術演算命令	166
13.1.1	加 算 命 令	166
13.1.2	減 算 命 令	168
13.1.3	補 数 命 令	169
13.1.4	多倍精度演算命令	169
13.1.5	ク リ ア 命 令	171
13.1.6	符号拡張命令	171
13.1.7	テ ス ト 命 令	172
13.1.8	比 較 命 令	172
13.1.9	乗 算 命 令	173
13.1.10	除 算 命 令	176
13.2	論理演算命令	178
13.3	2進化10進数演算命令	180
13.4	テストアンドセット命令	182
14	桁移動, ビット操作命令	184
14.1	論理形桁移動 (論理シフト) 命令	184
14.2	算術形桁移動 (算術シフト) 命令	187
14.3	循環形桁送り (ローテート) 命令	190

14.4	ビット操作命令	192
15	プログラム制御命令	195
15.1	分 岐 命 令	195
15.2	サブルーチン操作命令	198
15.3	引数の受渡し方法	200
15.3.1	値 呼 び	200
15.3.2	番 地 呼 び	202
15.3.3	アドレスレジスタをポインタとして渡す方法	203
15.4	LINK/UNLK 命令	204
15.5	条件セット命令	207
16	システム制御命令	208
16.1	トラップ発生命令	208
16.2	コンディションコード操作命令	210
16.3	その他の命令	212
17	特 権 命 令	215
18	例外処理後のシステム プログラム	220
18.1	リセット例外処理後のシステム プログラム	220
18.2	割込み例外処理後のシステム プログラム	221
18.3	トラップ例外処理後のシステム プログラム	222
18.4	不当命令および未実装命令例外処理後のシステム プログラム	222
18.5	特権違反例外処理後のシステム プログラム	222
18.6	トレース例外処理後のシステム プログラム	223
18.7	バスエラー例外処理後のシステム プログラム	223
18.8	アドレスエラー例外処理後のシステム プログラム	223
19	プログラム例	225
19.1	2 進化 10 進数の真数変換	225
19.2	1 の個数の数え上げ	226

19.3	文字列の一致判定	227
19.4	ASCII から 2 進数への変換	228
19.5	パリティの生成	229
III 68000 のサポート システム		
20	68000 マイクロコンピュータ システム	232
20.1	H 680 SBC	232
20.2	H 680 TR 01	235
21	68000 マイクロコンピュータ開発支援装置	238
21.1	H 680 SD 300 のハードウェアの概要	238
21.1.1	システム構成	238
21.1.2	システムの機能概要	239
21.1.3	システムの拡張性	241
21.2	ソフトウェア構成	241
21.3	HD 680 SD 300 の使い方	243
21.3.1	入出力機器の指定	243
21.3.2	システムの動作モード	244
21.3.3	ソフトウェアの開発手順	244
22	サポートソフトウェア	247
22.1	FDOS (Floppy Disk Operating System)	248
22.2	EMS (Executive Monitor System)	250
22.3	CRT エディタ	250
22.4	リンケージエディタ	251
22.5	マクロアセンブラ	252
22.6	FORTRAN	252
22.7	PASCAL	253
22.8	スーパ PL/H	254
付	録	257
索	引	267



# I 68000 のハードウェア

# 1 ま え が き

## 1.1 マイクロコンピュータの進歩

マイクロコンピュータ 4004 が発表されてから、すでに 10 年が過ぎた。この間、LSI 技術の進歩、特に MOS-LSI の高集積化、高性能化はマイクロコンピュータの機能、性能を大きく発展させた。図 1.1 は 10 年間のマイクロコンピュータの進歩を、1 個のチップ上に集積化された回路規模すなわち MOS トランジスタの数で示したものである。初期のマイクロコンピュータ 4004 は、アルミゲートの P チャネル構造の MOS-LSI 技術で実現されていた。この LSI 技術は、集積度も低く、トランジスタの動作速度も遅かったので、内部の演算データのビット幅を 4 ビットとして回路規模を小形化し、また命令実行時間も数十  $\mu\text{s}$  という低性能なマイクロコンピュータであった。

マイクロコンピュータの性能を飛躍的に改善するきっかけとなったのは、シリコンゲー

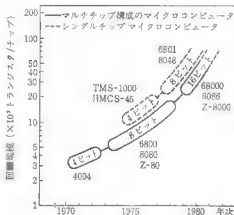


図 1.1 マイクロコンピュータの進歩

トの N チャネル構造 MOS-LSI 技術の出現であった。この技術により、内部演算データのビット幅が 8 ビット、命令実行時間が数  $\mu\text{s}$  というマイクロコンピュータが実現できた。6800、8080、Z-80 などが代表的な機種として広く用いられている。

一方この時期に、演算回路、メモリ、入出力回路を 1 個のチップ上に含んだ 4 ビットシングルチップマイクロコンピュータが現

れてきた。このマイクロコンピュータは、演算性能は比較的低いが、チップ上に多くの機能を盛り込んだLSIである。このようにマイクロコンピュータは、6800のような高性能マルチチップマイクロコンピュータとTMS-1000のような多機能シングルチップマイクロコンピュータの2極化の傾向が現れてきた。これらは産業界のあらゆる分野で使用され、マイクロコンピュータの普及の礎を築いたともいえる。

LSI技術の最近の進歩により、LSI内部の配線の最小幅は $3\mu\text{m}$ にまで微細化できるようになった。その結果さらに数倍～十数倍の集積度、スピードの向上を図れるようになった。このようなLSI技術を用いることによって16ビットの演算データ幅をもち、性能、規模で従来のミニコンピュータに匹敵するようなマイクロコンピュータが実現できるようになった。すでに世の中で使用されている16ビットマイクロコンピュータの代表的なものとしては、68000、8086、Z8000があげられる。

16ビットマイクロコンピュータは、性能、回路規模のみでなく、コンピュータアーキテクチャも、8ビット時代のものに比べ大きく進歩している。

これらの特徴のいくつかを以下に示す。

**a. メモリアドレス空間の拡大** 1メガバイト以上のメモリアドレス空間をもっている。その結果、より複雑、大規模なプログラムが搭載され、主に高級言語によるプログラム開発が行われるようになる。

#### **b. パフォーマンスの向上**

- (1) データ転送、演算の高速化のために、内部データ処理は最大32ビット単位で行われる。
- (2) クロックの高速化とパイプライン方式などの先行制御を行っている。
- (3) 乗算、除算、ストリング処理、ビット処理などの多機能命令の強化を図っている。
- (4) アドレス形式の種類の増加と多機能化により、命令のスループットを向上させている。
- (5) レジスタおよびスタックの機能の向上、数の増加により、処理効率を向上させている。

#### **c. アーキテクチャの向上**

- (1) マイクロプログラミング制御方式、モジュール構造化等の設計技術を導入している。
- (2) マルチプロセッサシステムを可能とするような命令、制御信号などを設けてい

る。

- (3) より大規模なシステムを効率良く動作させるために、非同期バスインターフェイスをもっている。

## 1.2 68000 の 概 要

16ビットマイクロコンピュータ68000は、 $3\mu\text{m}$ 加工のシリコンゲートNチャネルMOS技術を用いて実現され、 $6.44\text{mm} \times 7.26\text{mm}$ のシリコンチップ上に約68000個のトランジスタを含んでいる。初期の8ビットマイクロコンピュータ6800と比べると、その性能、回路規模は図1.2に示すように大幅に向上している。性能では約10倍、回路規模では約15倍となっている。さらにLSI技術の進歩を十分に活用することによって、将来に向けた新しいマイクロコンピュータアーキテクチャの導入を図っている。命令体系には、8ビットマイクロコンピュータ6800との共通性はない。その結果、旧LSI技術によって実現された8ビットマイクロコンピュータのアーキテクチャにとらわれることなく、斬新なアーキテクチャを導入することが可能となった。68000は、最新の半導体技術、回路技術、計算機技術を駆使し、最高の性能と柔軟性、信頼性、使いやすさを追求して得られたマイクロコンピュータといえる。さらに、それは将来の32ビットマイクロコンピュータへの発展を考慮した構造をもつ16ビットマイクロコンピュータである。その特徴の概要は次のとおりである。

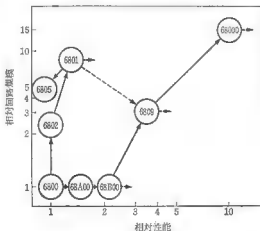


図 1.2 マイクロプロセッサの性能比較

- (1) 一貫性と規則性をもったアーキテクチャ
- (2) 多機能、高性能なプロセッサ構造
- (3) 効率的なプログラミング手法のサポート
- (4) ソフトウェアの信頼性向上
- (5) 高機能な例外処理
- (6) マルチプロセッサ用機能
- (7) 多機能インターフェイス

本書はⅢ編に分れている。Ⅰ編では、68000 のハードウェアについて述べる。そこでは、アーキテクチャ、インターフェイス信号、アドレス方式、命令、例外処理、周辺デバイスとのインターフェイス、68000 周辺 LSI について記述している。ここで使用している用語等は、極力モトローラ社あるいは日立製作所の説明書と同じになるように努めた。

Ⅱ編では、68000 のソフトウェアについて述べる。そこでは、アセンブラ、各命令グループの詳細な説明、割込み処理などを述べる。各命令の説明時に、必要に応じて短いプログラムの例をあげているが、最後に命令の使い方をより理解してもらうために、6 種のプログラム例を記載した。

Ⅲ編はサポートシステムである。ハードウェアおよびソフトウェアのサポートシステムについて述べる。紙面の都合で、詳細な説明は割愛した。ハードウェアサポートシステムは日立製作所の製品について紹介する。ソフトウェアサポートシステムも、日立製作所から提供されているものを紹介した。16 ビットマイクロコンピュータシステムにおいて、今後益々重要となるオペレーティングシステムについては、本マイクロコンピュータシリーズの別版として発刊されることを期待して、本書ではふれていない。

付録として、アドレス形式、および命令一覧表を載せ、本文理解の一助となるよう配慮してある。

## 2 68000 のアーキテクチャ

マイクロプロセッサ 68000 は、最新の VLSI (Very Large Scale Integrated Circuit) 技術と計算機技術により、高密度、高速性、高いフレキシビリティ、信頼性および使いやすさを実現した高性能 16 ビット マイクロプロセッサである。そのアーキテクチャは、マイクロプログラム制御方式の導入による斬新な LSI 構造を有するとともに、将来の 32 ビット マイクロプロセッサへの展開を考慮したものとなっている。

### 2.1 アーキテクチャの特徴

本節では、68000 MPU (Micro-Processing Unit) のアーキテクチャの特徴について簡単にふれるが、その詳細説明は、本章の次節以後および 3 章以後で行うこととする。

68000 アーキテクチャの特徴は次のとおりである。

(1) レジスタが多く、使いやすい。

32 ビットの多機能レジスタを 17 個もつ。そのうち 8 個はデータ用、9 個はアドレス用である。9 個のアドレス用のレジスタのうち 2 個はシステム スタック ポインタとして使用する。

(2) アドレス空間が広い。

16 メガバイトの物理 (フィジカル) アドレス空間をもつ<sup>†</sup>。さらに 64 メガバイトまでアドレス空間を広げる手段を有する。

(3) 命令は、レギュラな体系で構成され、機能が豊富である。

56 種の基本命令をもち、高級言語をサポートする命令を含む。

---

<sup>†</sup> 物理アドレスは、ハードウェアで定まるアドレスと定義する。これに対し論理アドレスは、ソフトウェア上で定義しているアドレスである。

(4) アドレス形式が強力である。

6種の基本モードがあり、そのなかに計14種のバリエーションをもつ。

(5) 種々のタイプのデータの演算が可能である。

1, 4, 8, 16, 32各ビット長のデータを扱える。

(6) 強力なベクタ割込み機能をもつ。

(7) マルチプロセッサシステム用のハードウェア、ソフトウェア制御機能をもつ。

(8) スループットの高いバスインターフェイスをもつ。

同期および非同期方式のバスインターフェイスをもつ。アドレスバスは24ビット、データバスは16ビットで構成されておりマルチプレックスされていない。また8ビットマイクロプロセッサの周辺LSIと直結できる。

(9) システムの信頼性向上、およびテスト効率向上を図る機能をもつ。

プログラム実行状態をスーパーバイザ状態とユーザ状態に分け、システムの信頼性向上を図っている。またトラップ、トレース機能を有し、システムのテスト効率を向上させている。

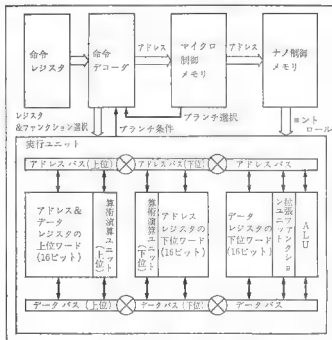
(10) メモリマップドI/O方式である。

周辺LSIなどに対しても、すべてのメモリアクセス命令を有効に使用できる。

以上のように、68000はミニコンピュータおよび大形コンピュータのアーキテクチャの設計思想を取り入れており、広い分野への応用を可能とするような、フレキシブルで、汎用的な構造を有している。

## 2.2 プロセッサの内部構造

68000は、以下に述べるようなマイクロプログラム制御方式を取り込んだ新しいタイプのVLSIアーキテクチャをもっている。図2.1に示すように、内部は大きく実行ユニットと命令制御部に分けられる。実行ユニットは、レジスタと演算回路で構成されている。これは、データおよびアドレスの演算を行う部分である。実行ユニットでは、アドレスおよびデータ用の2個の16ビット幅のバスで各部を結合している。各バスごとに並列にデータ転送ができるので、LSI内部での各レジスタ、演算回路間のデータ転送、演算の効率が向上する。バス構造のもう1つの特徴は、各バス上の2箇所にスイッチがある点である。これらのスイッチを制御することにより、互いに独立したセクションとしてバスを分離できる。分離された各セクション毎に上位ワード、下位ワードの転送を同時に行うことがで



ALU: 算術・論理演算ユニット

図 2.1 68000 の内部構造

きるため、内部のデータ転送効率を向上させることができる。さらに、3 個の演算制御ユニットでは、アドレス演算およびデータ演算が並列に行われる。

特に拡張ファンクションユニットは、32 ビット演算のためのシフト機能、ビット操作機能などの特別な論理操作を行い、ALU (Arithmetic Logic Unit) を補完する。以上の実行ユニットに隣接して、以下に説明する制御用の各ブロックが配置されている。

68000 の制御は、前述したようにマイクロプログラム制御方式を採用している。マイクロプログラム制御方式は、規則的なメモリ構造を利用して複雑な制御回路を実現できることが特徴である。また VLSI チップの開発時に、マイクロプログラム用メモリの書換えにより制御シーケンスが容易に設定でき、フレキシビリティに富むため、複雑な VLSI チップの設計法としてよく使われている。次に、このマイクロプログラムの動作を簡単に説明する。

オペレーションコードは LSI チップの外部のメモリから入力され命令レジスタに取り



込まれた後、図2.1に示されているように、命令デコーダで解読される。その結果、実行ユニットへはレジスタおよびファンクションの選択信号が出力され、マイクロ制御メモリへはマイクロ命令シーケンス起動のためのアドレスが出力される。68000のマイクロプログラム制御方式の特徴は、1個のLSIチップに全機能を効率良く集積化するために、2レベルの制御構造を採用し制御用メモリのサイズの小型化を図っていることである。第1のレベルは、ビット幅の短いマイクロ制御命令とよばれる命令形式で、複雑なブランチ能力により、マイクロプログラムのシーケンスを制御する。第2のレベルは、ビット幅の広いナノ制御命令とよばれる命令形式で、実行ユニットの動作を制御する。これらの制御シーケンスは、図2.1に示すように、それぞれ別のメモリ (ROM: Read Only Memory) に記憶されている。LSI 外部から取り込んだ命令によりマイクロ制御命令のシーケンスが起動されると、まずマイクロ制御命令からナノ制御メモリのアドレスが出力される。その出力アドレスによりナノ制御命令が読み出される。このナノ制御命令が直接に実行ユニットを制御することとなる。

## 2.3 レジスタの構成

図2.2は、68000のレジスタ構成を示す。68000のレジスタは8個のデータレジスタ



図 2.2 レジスタ構造

(D0~D7), 7個のアドレスレジスタ (A0~A6), 2個のシステムスタックポインタ, 1個のプログラムカウンタおよび1個のステータスレジスタで構成される。

68000は、構造的に統一性の高い16ビットのマイクロプロセッサである。特にレジスタ、データに関して次のような特徴をもつ。

(1) データレジスタは完全に対称である。すなわちどのデータレジスタも、オペレーションのオペランド(許されるアドレス形式内で)となり得る。

(2) すべてのデータレジスタとメモリ内のデータは、ビット(1ビット)、ディジット(4ビット)、バイト(8ビット)、ワード(16ビット)およびロングワード(32ビット)の演算が可能である。

(3) バイト、ワード、ロングワードの各オペランドは、整数演算を受ける。オペランドサイズは命令によって指定される。

(4) すべてのアドレスレジスタは、データをアドレッシングするために同等に使うことができる。

(5) アドレスレジスタのデータは、ワード、ロングワードのアドレス演算ができる。各レジスタ群について、以下に説明する。

### 2.3.1 データレジスタ (D0~D7)

8個のデータレジスタはそれぞれ32ビットの構成である。オペランドの指定により、これらをバイトレジスタ、ワードレジスタ、ロングワードレジスタとして扱うことができる。

図2.2に示すように、バイトの場合は下位8ビット、ワードの場合は下位16ビット、ロングワードの場合は32ビット全体を用いる。LSB (Least Significant Bit; 最下位ビット) はビット番号0、MSB (Most Significant Bit; 最上位ビット) はビット番号31として表される。データレジスタがバイトおよびワードオペレーションのデスティネーションオペランドとして用いられるときは、下位部のみ変化し、残りの上位部は変化しない。

### 2.3.2 アドレスレジスタ (A0~A6), システムスタックポインタ (A7)

アドレスレジスタとシステムスタックポインタは32ビット長で構成されている。これらはバイトサイズで用いることはできない。したがってアドレスレジスタがオペレーシ

† MOVEQ, MOVEM 命令は例外

レジスタのソースオペランドとして用いられるときは、オペレーションサイズに応じて下位のワードか、ロングワードがオペランドとして用いられる。アドレスレジスタがデスティネーションオペランドとして用いられるときは、オペレーションサイズに関係なく、そのレジスタの全ビットが影響を受ける。オペレーションサイズがワードならば、オペランドは演算実行前に、その15ビット目（符号ビット）が16ビットから31ビットまでの上位の全ビットに自動的にコピーされ演算される（符号拡張：2.1節で述べる）。

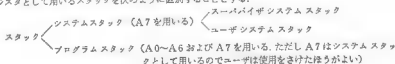
なおデータレジスタ、アドレスレジスタは、共にインデックスレジスタとして用いることも可能である。

A7はアドレスレジスタとして用いられるのみでなく、システムスタックポインタにもなっている。ここでスタック<sup>†</sup>とは、データの後入れ、先出し（Last-In-First-Out）リスト構造のレジスタ群（メモリ）のことである。A7は割込みのときなど、システムスタックポインタとしてプロセッサが自動的に内容を変化させる場合があるので、プログラムスタックポインタとして使用するときには注意しなければならない。A7をシステムスタックポインタとして用いるときには、アドレスを自動的に増加して、新しいメモリアドレスにデータ（戻り番地、例外処理時の退避データ等）を書き込んだり、自動的に減少して、すでにデータが書き込まれているアドレスからデータを読み出したりする機能をもつ。

スタックポインタは、図2.2のA7に示すように、ユーザ状態のときと、スーパーバイザ状態のときで、自動的に切り換えて使用される。すなわちA7のレジスタは、ユーザプログラム状態のときは、ユーザシステムスタックポインタ（USP）となり、スーパーバイザプログラム状態のときは、スーパーバイザシステムスタックポインタ（SSP）となる。68000がスーパーバイザ状態であるとき、A7のレジスタがスーパーバイザシステムスタックポインタとなっているから、ユーザシステムスタックポインタはアドレスレジスタとして参照することはできない。また逆も同様である。

† 68000のマニュアル等では、システムスタックとユーザスタックという名称で2つのタイプのスタックを定義している。後者はアドレスレジスタA0～A6およびA7を用い、命令のアドレスレジスタ間接モードのポストインクリメント（実行後アドレスレジスタに1を加える）およびプリデクリメント（実行前にアドレスレジスタから1を減ずる）の各操作を行うことにより、プログラムのスタックを構成するものである。これに関しては12章でプログラムの一例として示す。

本書ではA7をスタックポインタとして用いるスタックとA0～A6およびA7をアドレスレジスタとして用いるスタックを次のように区別することとする。



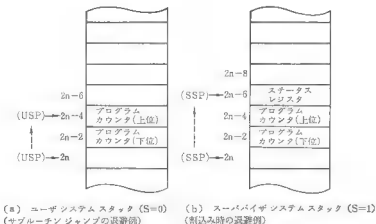


図 2.3 システムスタック

これらのシステムスタックポイントにより、図 2.3 に示すようなユーザシステムスタック ( $S=0$  のとき) とスーパーバイザシステムスタック ( $S=1$  のとき) を実現できる。

図 2.3 に示すように、これら 2 つのシステムスタックはメモリアドレスの大きい方から小さい方へ順に使用される。すなわちシステムスタックにデータがプッシュされるとスタックポイントは減算され、システムスタックよりデータが取り出されるとスタックポイントは増加する。

サブルーチン呼出しの際は、プログラムカウンタがアクティブなシステムスタックに退避され、リターン時にアクティブなシステムスタックから復元される。一方トラップ処理、割込み処理などの例外処理が起動されるとスーパーバイザ状態となり、プログラムカウンタ、ステータスレジスタなどをスーパーバイザシステムスタックへ退避する。

システムスタックへの退避情報についての詳細は 8 章にて説明する。

システムスタック上のデータ配列を適切に行うために、システムスタックに対するデータエントリのアドレスはワードの境界に限定され、データは常にワードの境界単位で格納される。

†  $S$  は後述するステータスレジスタ内のフラグである。

### 2.3.3 プログラムカウンタ (PC)

68000 は 32 ビットのプログラムカウンタをもつが、現在はその下位 24 ビットのみを LSI チップ外に出力している。したがって生成できるアドレスは、16 進表示で \$000000 から \$FFFFFF (10 進表示では、0 から 16 777 215) の範囲となる。命令長がワード単位であるため命令は偶番地で示されるアドレスに格納されなければならない。このためプログラムカウンタには必ず偶数アドレスを設定しておく必要がある。奇数アドレスを設定した場合はアドレスエラーが発生する。このプログラムカウンタの 1 ビットから 23 ビットがチップ外に出力され、A1~A23 のアドレスバスとなっている。

### 2.3.4 ステータスレジスタ

ステータスレジスタは図 2.4 に示すように、16 ビットで構成され、ユーザバイトとシステムバイトに分けられている。図 2.4 にステータスレジスタのビット構成を示す。

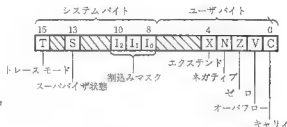


図 2.4 ステータスレジスタ

**a. ユーザバイト** ユーザバイトには、キャリイ (C)、オーバーフロー (V)、ゼロ (Z)、ネガティブ (N)、エクステンド (X) のコンディションコードフラグが割り当てられている。

**b. システムバイト** システムバイトは、次の 3 種のフラグで構成される。

(i) 割込みマスク (I0~I2) 3 ビットで、現在実行中のプログラムの優先順位（プライオリティレベル：0~7）を指定する。ここで示されたレベルと同じかまたは低いレベルの割込み要求は受け付けられない。ただしレベル 7 の割込みは常に受け付けられる。

(ii) スーパーバイザ状態フラグ (S) 実行プログラムがスーパーバイザ状態 (S=1) であるか、ユーザ状態 (S=0) であるかを示す。

(iii) トレースモードフラグ (T) 本ビットが“1”にセットされていると、命令実行ごとにトレース例外処理を行う。すなわち命令を実行するたびにスーパーバイザ状態に入

り、ユーザのトレース サービス ルーチンへジャンプする。

## 2.4 データの構成

### 2.4.1 メモリ内のデータの構成

68000 のシステムにおけるメモリ内のワードおよびバイト構成を図 2.5 に示す。1 ワードを構成する 2 バイトのうち、上位バイトのアドレスはワードデータのアドレスと同じ偶数アドレスとなっている。下位バイトはワードデータのアドレスより 1 だけ大きい奇数アドレスになっている。これらバイトデータは個別にアクセスすることができる。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ワード								0 0 0 0 0 0							
バイト								0 0 0 0 0 0							
ワード								0 0 0 0 0 2							
バイト								0 0 0 0 0 2							
ワード								0 0 0 0 0 4							
バイト								0 0 0 0 0 4							
ワード								0 0 0 0 0 5							
バイト								0 0 0 0 0 5							
ワード								F F F F F C							
バイト								F F F F F C							
ワード								F F F F F E							
バイト								F F F F F E							

図 2.5 メモリのワード/バイト構成

命令 (1 ワード以上で構成される) および複数バイト (ワード、ロングワード) データは、ワードの境界ごと (偶数アドレス) にアクセスされる。ロングワードデータが偶数アドレス  $n$  に格納されていると、次のデータはアドレス  $n+4$  に格納されることとなる。

68000 は、ビットデータおよび 8 ビット、16 ビット、32 ビットの整数データ、ならびに 32 ビットのアドレスデータ、2 進 10 進数 (BCD: Binary Coded Decimal) 表現のデータの演算を行う。これら各形式のデータがメモリ内に格納されときのアドレス配置を図 2.6 に示す。図中でハッチした部分が命令によって演算される単位である。2 進 10 進数データは 1 バイトの中に 2 デジット (1 デジット: 4 ビットで構成される) を含むように構成されている。10 進演算はバイト単位で行われるので、2 デジット分を同時に演算する。

## ビットデータ

- 1 バイト = 8 ビット



## 整数データ

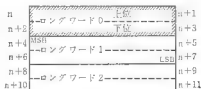
- 1 バイト = 8 ビット



- 1 ワード = 16 ビット

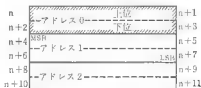


- 1 ロングワード = 32 ビット



## アドレス

- 1 アドレス = 32 ビット



## 10進数データ



MSB: Most Significant Bit, LSB: Least Significant Bit

MSD: Most Significant Digit, LSD: Least Significant Digit

図 2.6 メモリ内の演算データの構成

## 2.4.2 整数データの表現形式

68000 では 2 値数, 2 進化 10 進数および 2 の補数表現による 2 進数のデータ形式を扱う。ここで 2 の補数表現について説明する。バイト, ワード, ロングワードの整数を 2 の補数 (2's Complement) 表現で表すと図 2.7 のようになる。各サイズのデータは, 最上位ビットに符号ビットを配置している。したがってバイトデータは  $-128 (\$80) \sim +127 (\$7F)$ , ワードデータは  $-32768 (\$8000) \sim +32767 (\$7FFF)$ , ロングワードは  $-2147483648 (\$80000000) \sim +2147483647 (\$7FFFFFFF)$  の範囲のデータを表現できる。

68000 では, バイトからワードまたはロングワードへ, ワードからロングワードへとデ

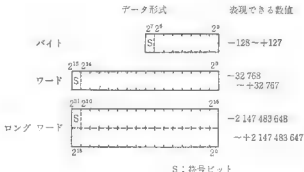


図 2.7 整数データの表現形式 (2の補数表現の場合)

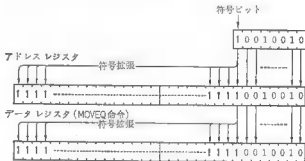


図 2.8 符号拡張 (バイト→ロングワードの例)

ータサイズを拡張できる。このときの整数データの符号の統一を図るために、符号拡張機能をもっている。

データサイズの拡張時における符号拡張機能について図2.8に示す。この図ではバイトデータからロングワードデータへの拡張の例を示す。符号拡張を行うには、ソースデータの符号ビットの内容が、そのビット番号より大きなビット位置にコピーされる。符号ビット以外の各ビットの内容はそれぞれの対応したビット番号の位置へ移る。この符号拡張は各データの演算の前に実行される。

アドレスレジスタがデスティネーションレジスタとして指定される場合は、常に符号拡張が行われる。

データレジスタがデスティネーションレジスタとして指定される場合は、MOVEQおよびMOVEMの命令を実行したときのみ符号拡張が行われる。



## 2.5 基本システム構成と動作

### 2.5.1 入出力インターフェイスの構成と特徴

68000 の LSI チップは、図 2.9 に示すような 64 ピンのデュアルインライン (Dual In Line) 形のパッケージに搭載されている。本章では入出力インターフェイスの構成の概要と特徴について述べる。

**a. 非同同期パラレルバス** 非同同期バス制御ラインには、アドレスストロブ ( $\overline{AS}$ )、リード/ライト制御 ( $R/W$ )、上位データストロブ ( $\overline{UDS}$ )、下位データストロブ ( $\overline{LDS}$ ) およびデータアクセスノレージ ( $\overline{DTACK}$ ) がある。これらの制御信号によって、メモリ、周辺 LSI などデータアクセス時間の異なる外部デバイスとのデータ転送効率を向上させる。68000 MPU はメモリ、周辺 LSI との間でデータを転送を行うときに、相手からデータ転送が完了することを知らせる信号 ( $\overline{DTACK}$ ) を入力することによって、データ転送サイクルを終了させる。

**b. マルチプロセッサ用のバスインターフェイス** 同一バス上にバスを支配できるデバイス (バスマスタ) が複数存在する場合 (たとえば複数個の MPU または DMAC)、そのバスの支配権のやりとりが、バス要求 ( $\overline{BR}$ )、バスグラント ( $\overline{BG}$ )、

バスグラントアノレージ ( $\overline{BGACK}$ ) 信号によって、ハンドシェーキング方式で行われる。この一連の動作がバスアービトレーションとよばれる。

**c. 非マルチプレックスバス** マイクロプロセッサのような LSI では、パッケージのピン数の制約から、1 本の入出力信号線に複数の機能をもたせる場合がある。特にメモリのアドレス空間を広くとるために、アドレスとデータを時分割で同一バス上にのせる方法が採用されることがある。しかしこの方法ではデータの転送効率を低下させる欠点があるため、68000 では 23 ビットのアドレスバス A1~A23 と  $\overline{UDS}$ 、 $\overline{LDS}$  信号および 16 ビ

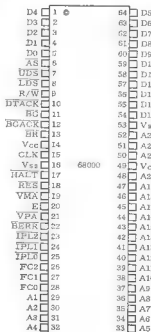


図 2.9 ピン配置

ットの双方向データバス D0~D15 を別々に設けている。これによって高速なバスサイクルを実現している。

**d. メモリマップド I/O** メモリ、周辺 LSI などのすべての外部デバイスを単一バス上に結合し、コミュニケーションを行う。すなわち同一メモリアドレス空間に、メモリ、周辺 LSI のアドレスを割り付ける。これによって周辺 LSI に対しても、68000 のすべての命令、アドレス形式が有効に利用できる。

**e. 優先割込み構造** 68000 には 3 本 ( $\overline{\text{IPL0}}$ ,  $\overline{\text{IPL1}}$ ,  $\overline{\text{IPL2}}$ ) の割込み制御入力がある。これは割込みを要求するデバイスからの優先レベルをコード化した入力信号である。レベル 0 は割込み要求のないことを示し、レベル 7 が最優先のノンマスクابل割込み要求であることを示す。

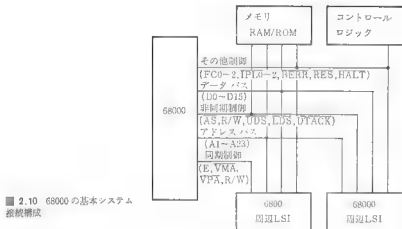
この入力割込みレベルが、内部のステータスレジスタ<sup>†</sup>に格納されている割込みマスクレベルと比較され、それ以下のときは割込みは受け付けられない。

**f. 8 ビットマイクロコンピュータ 6800 ファミリの LSI と直結できるバスインターフェイス** 68000 は上記非同期バスインターフェイスのほかに、同期式のバスインターフェイス制御信号線をもつ。図 2.9 のイネーブル (E), バリッドメモリアドレス ( $\overline{\text{VMA}}$ ), バリッドペリフェラルアドレス ( $\overline{\text{VPA}}$ ) およびリード/ライト制御 (R/W) を同期式インターフェイスのために用いる。これによって 68000 のシステムに同期式のバスインターフェイスをもつ 8 ビットマイクロプロセッサ 6800 の周辺 LSI (PIA, PTM, FDC, CRTC, ACIA, SSDA, ADLC, GPIA 等) を利用することができる。

**g. メモリマネージメント機能** 3 本のプロセッサステータス信号線 (FC0, FC1, FC2) を介して、プロセッサのプログラム実行状態を知ることができる。これを用いてメモリ空間にユーザデータ、ユーザプログラム、スーパーバイザデータ、スーパーバイザプログラム領域を別々に設けることができる。その結果アドレス空間を 64 メガバイト (16 メガバイト 4 セグメント) にまで拡張できる。これらの信号線を使うことによって、メモリ管理ユニット MMU (Memory Management Unit) によるメモリ保護、アドレスマッピング等を行うことができる。

**h. バスエラー入力信号** 入力信号  $\overline{\text{BERR}}$  はバスエラー発生を示す入力で、不応答デバイス、不当メモリアクセス、割込みベクタ番号取込み不良などを 68000 に知らせる

<sup>†</sup> 割込み制御入力は負論理であるが、内部のステータスレジスタの情報は正論理で表されている。



のに有効である。この信号によってバスサイクルを終了させ68000は例外処理の実行に移す。図2.10は、68000の基本システムの接続構成を示す。

## 2.5.2 基本動作サイクル

68000のサイクルタイムは次のように分けられる。

- クロックサイクル** 入力クロックの周期である。そのクロックパルスの立上りエッジから次のクロックパルスの立上りエッジまでの時間である。
- バスサイクル** バイト、ワードデータの読出し、書込みを行うのに必要なタイミングシーケンス全体である。4クロックサイクルで最小バスサイクルが構成される。非同期バスインターフェイスで  $\overline{DTACK}$  信号が遅れて入力されるときは、本バスサイクルは  $4+n$  クロックの形で延長される。
- 命令サイクル** 1命令を実行するのに必要なタイミングシーケンスである。一般に複数のバスサイクルによって構成される。各命令の実行時間は、上記クロックサイクルの数で示される。

## 2.5.3 命令プリフェッチ機能

68000は高速性を実現するために命令プリフェッチ（命令の先取り）とよばれる機能を持つ。この機能はある命令を実行中に次の命令をMPU中に取り込むもので、命令フェッチに要する時間を見掛上節約できるという利点がある。68000はこの機能を実現するため

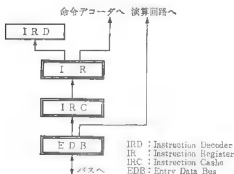


図 2.11 命令プリフェッチ用レジスタ群

に図 2.11 に示すような 3 本のレジスタを有している。現在実行中の命令コードは IR (Instruction Register) にセットされ、命令がデコードされたのち、IRD (Instruction Decoder) に移される。命令が IRD に移されると次に実行される命令が IRC から IR に移されデコードの開始を待つ。原則として IR、IRC (Instruction Cache) には次に実行され

る命令の第 1 ワード目が各々セットされる。命令の第 2 ワード目からの拡張部分は EDB (Entry Data Bus) にセットされる。このように、プリフェッチ用の特別なレジスタを有することにより次に実行される命令の全体または一部を先取りし、実効的に命令フェッチ時間を節約し得るだけでなく、IRD を設けることにより次の命令のデコードを早期に開始し高速化を図ることが可能となる。

シングルワードの分岐命令 (Bcc, DBcc, JMP 命令等) を実行したとき、分岐が行われる場合は、プログラムの流れが飛ぶために、あらかじめフェッチ済の次のアドレスの命令ワードは不要となる。そのとき、新たに 2 ワードの命令をフェッチするような制御が行われる。68000 では前の命令実行終了までに 2 ワードがプリフェッチされるため、アドレス空間の最後の 1 ワードにシングルワードの分岐命令などを配置しておく、前の命令の実行中に行われる命令フェッチ動作でアドレスエラーを発生するので注意が必要である。

#### 2.5.4 基本命令シーケンス

図 2.12 に 68000 の基本命令シーケンスの一例を示す。前項で述べたプリフェッチ機能を有しているために、各命令の実行時には、引き続いて行われる命令列の 2 ワード分がメモリから MPU 内に取り込まれる。図にハッチで示したバスサイクルは MPU の演算結果のワードデータをメモリへ転送するサイクルである。条件ブランチ命令 (Bcc) を実行するときは、2 クロックの区間で条件判定を行う。この間はデータバスは使用されない。前節で述べたように分岐が行われるときは、前のサイクルでフェッチされた命令ワードは捨てられ、分岐先の新命令を 2 ワードフェッチする。

分岐命令で分岐したときは新たに命令を 2 ワードフェッチするが、それ以外の通常命令

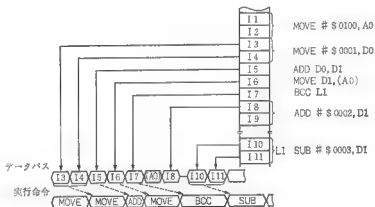


図 2.12 基本命令シーケンス

では、その命令自身のワード数分だけ新しい命令ワードをフェッチするようなシーケンスをもっている。

# 3

## インターフェイス信号とバスオペレーション

この章では68000のインターフェイス信号の機能について説明し、次に、それらの信号の基本動作であるバスオペレーションの種類とタイミングについて述べる。

### 3.1 インターフェイス信号

68000のインターフェイス信号は図3.1に示すようなグループに機能的分類をすることができる。以下に各機能グループごとに信号の説明を行う。

#### 3.1.1 アドレスバス、データバス

**α. アドレスバス (A1~A23) [出力]** 68000のアドレスバスは23ビットで構成され、8メガワード (16MByte) のデータを直接にアドレッシングすることができる。ア

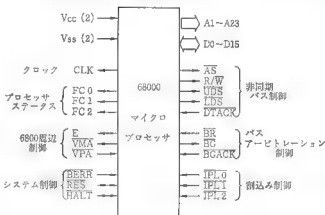


図 3.1 信号の機能グループによる分類

アドレスバスは単方向性のスリーステートバスで、割込みサイクルを除くバスオペレーション・サイクルに対してアドレスを供給する。ここで供給されるアドレスはワードデータ（16ビット）をアクセスするためのもので、バイトデータ（8ビット）をアクセスするためにはA0なるアドレス信号が必要である。68000におけるアドレス付けはバイトを単位として行われ、プロセッサ内部にはA0のアドレスが存在する。A0のアドレスは直接には外部に出力されていないが、プロセッサがバイトデータをアクセスするときにはA0の値に応じて上位データストロープ（ $\overline{\text{UDS}}$ ）または下位データストロープ（ $\overline{\text{LDS}}$ ）が出力される。これについては後の3.1.2項cで述べる。

アドレスバスは、割込みサイクルのときには別の意味をもつ。すなわち、割込みアノレッジサイクルではアドレスバスのA1, A2, A3からはどのレベルの割込みがサービスされているかについての情報が供給される。なお、アドレスバスA4~A23はこのときすべて“High”レベルにセットされる。

**b. データバス（D0~D15）[入出力]** データバスは16ビットの双方向性のスリーステートバスで、プロセッサが外部のデバイス（メモリや周辺装置）とデータの授受を行うための汎用のデータ通信路である。データ授受はワード長（16ビット）またはバイト長（8ビット）のいずれでも可能である。

データバスを介してプロセッサと外部デバイスの間で授受されるデータは、通常、プロセッサを動作させるための命令およびデータである。ただし割込みアノレッジサイクルのときだけは、外部デバイスからプロセッサに対して例外ベクタ番号が転送される（プロセッサはこのときにバス上に乗っているデータを例外ベクタ番号として処理する）。

### 3.1.2 非同期バス制御信号

以下の信号はプロセッサが外部デバイスとデータの授受を非同期的に行うための信号である。

**a. アドレスストロープ（ $\overline{\text{AS}}$ ）[出力]** アドレスストロープはアドレスバスに有効なアドレスが出力されていることを示す信号である。

**b. リード/ライト（R/ $\overline{\text{W}}$ ）[出力]** この信号はデータバス上のデータ転送がリードサイクルなのかライトサイクルなのかを定める。すなわち、R/ $\overline{\text{W}}$  = “High” のときリードサイクル、R/ $\overline{\text{W}}$  = “Low” のときライトサイクルである。

**c. 上位データストロープ、下位データストロープ（ $\overline{\text{UDS}}$ ,  $\overline{\text{LDS}}$ ）[出力]** データストロープ信号はプロセッサがデータバス上のデータを読み取るタイミングまたはデー

タバス上へデータを出力するタイミングを示すものである。データストロブ信号は本来は1本だけでもすむが、68000の場合は  $\overline{UDS}$ 、 $\overline{LDS}$  と2本用意することにより、メモリのワードだけでなくバイトごとのリード/ライトオペレーションが可能となっている。

表3.1に  $\overline{UDS}$ 、 $\overline{LDS}$  の働きを示す。この表からわかるように、 $\overline{UDS}$ 、 $\overline{LDS}$  がインアクティブ<sup>†</sup>であるかぎりリード/ライトオペレーションは行われぬ。また、ワードのリード/ライトオペレーションでは  $\overline{UDS}$ 、 $\overline{LDS}$  のいずれもがアサートされ、バイトのリード/ライトオペレーションでは内部のアドレス A0 の“Low”、“High”に対応して、それぞれ  $\overline{UDS}$ 、 $\overline{LDS}$  の一方だけがアサートされる。

表 3.1 データバスのデータストロブ制御

R/W	$\overline{UDS}$	$\overline{LDS}$	A0	D8~D15	D0~D7	オペレーション
—	High	High	—	**** <sup>†1</sup>	**** <sup>†1</sup>	No operation
High	Low	Low	—	D8~D15	D0~D7	ワードリード
High	High	Low	High	**** <sup>†1</sup>	D0~D7	下位バイトリード
High	Low	High	Low	D8~D15	**** <sup>†1</sup>	上位バイトリード
Low	Low	Low	—	D8~D15	D0~D7	ワードライト
Low	High	Low	High	D0~D7 <sup>†2</sup>	D0~D7	下位バイトライト
Low	Low	High	Low	D8~D15	D8~D15 <sup>†3</sup>	上位バイトライト

†1 \*\*\*\*は無効データ

†2 D8~D15 には D0~D7 と同じ内容が出力される。

†3 現状の68000では D0~D7 には D8~D15 と同じ内容が出力される。

d. データ転送アクノレッジ ( $\overline{DTACK}$ )【入力】 この信号は外部デバイスがデータをリードまたはライトすることが可能になったタイミングをプロセッサに知らせるための信号である。後で述べるようにプロセッサのリードまたはライトのバスサイクルは S0~S7 の8つの状態で構成され、プロセッサは S4 の後縁で外部から  $\overline{DTACK}$  がアサートされているか否かをチェックする。もし、このとき、 $\overline{DTACK}$  がアサートされていなければ、S4 と S5 の間に Sw (ウェイト状態) が挿入され、プロセッサは  $\overline{DTACK}$  がアサートされるまで待つことになる。すなわち、アクセスタイムの遅い外部デバイスも、そのアクセスタイムに応じて  $\overline{DTACK}$  を遅らせてアサートすれば、プロセッサとのデータ授受を確実に行うことができる。したがって、外部デバイスとプロセッサ間の非同期の

† 68000のマニュアルには、信号がアクティブまたは真 (true) であることを示すためにアサート (Assert) という言葉を用い、インアクティブ (休止状態) が偽 (False) であることを示すためにネゲート (Negate) という言葉を用いている。しかし本書では従来からの慣例からみて、アクティブ、インアクティブの方が、各信号の電圧の“Low”、“High”のいかんにかかわらず、実動作をよく表現していると考えられる場合は、後者の用語を用いた。



データ転送を可能にしているのは、この  $\overline{\text{DTACK}}$  信号である。

### 3.1.3 バスアービトレーション制御信号

以下の3つの信号は、プロセッサも含めて2つ以上のバスマスタになり得るデバイスが存在するときに、どのデバイスがバスマスタになるかを決定するのに必要な信号である。

a. **バス要求 (BR) [入力]** この信号は、外部デバイスがバスマスタになることをプロセッサに要求する信号である。この信号の入力端子には、バスマスタとなり得るすべてのデバイスからの信号のワイヤドオアがとられたものが入力される。

b. **バスグラント ( $\overline{\text{BG}}$ ) [出力]** この信号は、プロセッサが他のすべてのバスマスタとなり得るデバイスに対してバスの解放を行うことを示す信号である。バスの解放はそのとき実行中のバスサイクルが終了した時点で行われる。

c. **バスグラントアクノレッジ ( $\overline{\text{BGACK}}$ ) [入力]** この信号は、外部のデバイスがバスマスタになったことを示す入力信号である。外部デバイスは以下の4つの条件を検出するまではこの信号をアサートしてはならない。

- (1) バスグラント信号 ( $\overline{\text{BG}}$ ) を受信すること
- (2) アドレスストロブ信号 ( $\overline{\text{AS}}$ ) が休止状態(インアクティブ)であること。つまり、プロセッサがバスを使用していないこと
- (3) データ転送アクノレッジ信号 ( $\overline{\text{DTACK}}$ ) が休止状態であること。つまり、メモリまたは入出力デバイスがバスを使用していないこと
- (4) バスグラントアクノレッジ信号 ( $\overline{\text{BGACK}}$ ) が休止状態であること。すなわち、バスマスタ権をとっている外部デバイスが他に存在しないこと

### 3.1.4 割込み制御信号

**割込み優先レベル ( $\overline{\text{IPL0}}$ ,  $\overline{\text{IPL1}}$ ,  $\overline{\text{IPL2}}$ ) [入力]** これら3本の割込み制御信号は外部デバイスがプロセッサに対して割込み要求があることを示すために用いられる。外部の割込みは3ビットにエンコードして入力され、7レベルの割込みが可能である。割込みレベルは7が最も優先度が高く、1が最も優先度が低い。また0は割込みがないことを示す。ここで、 $\overline{\text{IPL0}}$  が最下位ビット (LSB)、 $\overline{\text{IPL2}}$  が最上位ビット (MSB) である。

### 3.1.5 システム制御信号

a. **バスエラー ( $\overline{\text{BERR}}$ ) [入力]** この入力信号は、現在実行中の命令サイクルで障

害が生じたことをプロセッサに知らせるための信号である。ここでいう障害とは次のようなものである。

- (1) デバイスが応答しないとき
- (2) 割込みベクトル番号をとらえられないとき
- (3) 不正なアクセス要求がメモリ管理ユニット (MMU) によって検出されたとき
- (4) 他に応用システムに依存するいろいろなエラーが発生したとき

**b. リセット ( $\overline{\text{RES}}$ )** [入出力] この双方向信号線は、外部のリセット信号に応じてプロセッサをリセットする働きをもっている。すなわち、この端子に "Low" レベルが入力されると、プロセッサはシステムイニシャライズシーケンスを開始する。また、プロセッサの内部で RESET 命令が実行されると、この端子よりリセット信号が出力される。すなわち、プロセッサは外部デバイスをリセットすることができる。ただしこの場合プロセッサ自身の状態はリセットされない。なお、プロセッサおよび外部デバイスを含めた全システムをリセットするには、リセットとホールド端子に同時に Low レベルを入力する必要がある。

**c. ホールド ( $\overline{\text{HALT}}$ )** [入出力] この双方向信号線へ外部から "Low" レベルを入力すると、プロセッサは現在のバスサイクルを完了した時点で停止する。プロセッサが停止すると、すべての制御信号は休止状態になり、データバスおよびアドレスバスはハイインピーダンス状態になる。また、プロセッサが2重バス障害のような状態になったときもプロセッサは停止し、このことを外界に知らせるため、この端子より "Low" レベルが出力される。

### 3.1.6 6800 周辺 LSI 制御信号

以下の3つの信号線は、68000 が6800 の周辺 LSI を使用することを可能にするために設けられたものである。

- a. イネーブル (E)** [出力] この信号は6800 周辺 LSI に共通なデータ転送の同期クロック信号である。このクロックの周期は68000 の10クロックサイクルに等しく6クロックが "Low"、4クロックが "High" という仕様になっている。
- b. バリッドベリフェラルアドレス ( $\overline{\text{VPA}}$ )** [入力] この信号はセレクトされたデバイスが6800 周辺 LSI であることをプロセッサに知らせるための信号である。プロセッサはこの信号を受けるとデータの転送をイネーブル信号 (E) に同期して行い、また、割込みをオートベクタ割込み (9章参照) で行うようになる。

e. バリッドメモリアドレス (VMA) [出力] この出力信号は、アドレスバス上に有効なアドレスが出力されていることを示す。同時に、VPA入力に対する応答であること、すなわち、データ転送がイネーブルに同期して行われることを示す。

表 3.2 ファンクションコードとサイクルタイプ

FC2	FC1	FC0	Cycle Type
Low	Low	Low	(Undefined Reserved)
Low	Low	High	User Data
Low	High	Low	User Program
Low	High	High	(Undefined Reserved)
High	Low	Low	(Undefined Reserved)
High	Low	High	Supervisor Data
High	High	Low	Supervisor Program
High	High	High	Interrupt Acknowledge

表 3.3 信号のまとめ

信 号 名	モニタ	入力/出力	アクティブ状態	スリープ スタート
アドレスバス	A1~A23	出力	High	○
データバス	D0~D16	入出力	High	○
アドレスストロブ	AS	出力	Low	○
リード/ライト	R/W	出力	Read: High Write: Low	○
上位データストロブ, 下位データストロブ	UDS, LDS	出力	Low	○
データ転送アノレシ	DTACK	入力	Low	—
バス要求	BR	入力	Low	—
バスグラント	BG	出力	Low	—
バスグラントアノレシ	BGACK	入力	Low	—
優先度レベル	IPL0, IPL1, IPL2	入力	Low	—
バスエラー	BERR	入力	Low	—
リセット	RES	入出力	Low	オープン ドレイン
ホールド	HALT	入出力	Low	オープン ドレイン
イネーブル	E	出力	High	—
バリッドメモリアドレス	VMA	出力	Low	○
バリッドペリフェラルアドレス	VPA	入力	Low	—
ファンクションコード	FC0, FC1, FC2	出力	High	○
クロック	CLK	入力	High	—
電源 (5V)	V <sub>CC</sub>	入力	—	—
接地 (0V)	V <sub>SS</sub>	入力	—	—

(注) ○印はスリープスタート信号であることを示す。

### 3.1.7 プロセッサステータス

**ファンクションコード (FC0, FC1, FC2)** このファンクションコード出力は、表3.2に示すようにプロセッサのプログラム実行状態ならびにメモリ等のアクセスの種類を示す。すなわち、ユーザモードかスーパーバイザモードか、プログラムフェッチサイクルかデータアクセスサイクルかが示される。さらに、これら3つの出力がすべて“High”レベルの場合はプロセッサが割込みを受け付けたこと（割込みアクトレッジ）を表す。

### 3.1.8 クロック (CLK)

クロックはプロセッサを動作させるための信号で、その最大周波数が4, 6, 8, 10, 12 MHzのMPUが用意されている。

### 3.1.9 信号のまとめ

表3.3に今まで述べてきた信号のまとめを示す。この表には、信号名とそのニモニク、その信号の入出力の区別、アクティブ状態が“High”レベルなのか“Low”レベルなのか記されている。さらに、この表の右端には、その信号がスリーステートになり得るかどうか示されている。

## 3.2 バス オペレーション

この節では68000のバスオペレーションについて説明する。バスオペレーションにはデータ転送、バスアービトレーション、バスエラー、およびホールド、リセット等のオペレーションがある。なお、割込み発生時のバスオペレーション、6800周辺LSIとのデータ転送オペレーション、また、DMA時のバスオペレーションについては8.9および10章にて説明する。

### 3.2.1 データ転送オペレーション

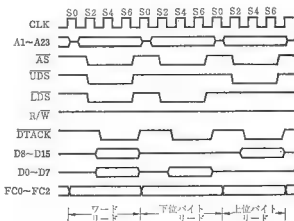
プロセッサと外部デバイスとの間で行うデータ転送はデータ転送オペレーションに従って行われる。このオペレーションでは、データバス(D0~D15)、アドレスバス(A1~A23)および非同期データバス制御信号( $\overline{AS}$ ,  $\overline{UDS}$ ,  $\overline{LDS}$ , R/W,  $\overline{DTACK}$ )が用いられる。また、そのとき実行されているバスオペレーションのタイプを示すためファンクシ

■ ノコード (FC0~FC2) が出力される。

データ転送オペレーションには、リードサイクル、ライトサイクルおよびリードモディファイライトサイクルの3つの基本オペレーションがある。以下、これら3つの基本オペレーションについて説明する。

■ リードサイクル リードサイクルはプロセッサが外部デバイスからデータを読み取るオペレーションである。プロセッサが実行すべき命令は外部メモリに格納されているので、命令フェッチもこのリードサイクルで行われる。実行中のリードサイクルが命令であるか、データであるかはファンクションコード (FC0~FC1) に出力表示される。

図3.2にリードサイクルのタイミングチャートを示す。ここでリードサイクルとしてワードリード、上位バイトリード、下位バイトリードの3つが示されているが、その違いは表3.1でも示したように、上位データストロブ ( $\overline{UDS}$ )、下位データストロブ ( $\overline{LDS}$ ) のアサートの仕方が違うだけである。ここではワードリードサイクルについてそのオペレーションを説明する。



■ 3.2 リードサイクルタイミング

S0 状態では、まず、R/W ラインがリード状態 (High レベル) にされ、ファンクションコードが FC0~FC2 へ送出される。S0 状態ではアドレスバスはまだハイインピーダンス状態である。次に S1 状態<sup>†</sup>になるとアドレスバスのハイインピーダンス状態が解除され、そこからアクセスすべきデバイスのアドレスが送出される。S2 状態ではアドレス

<sup>†</sup> S1 状態: S0 と S2 の間の CLK=Low レベルの状態である。本書ではクロックサイクルの奇数状態を省略している場合がある。

ストロブ ( $\overline{AS}$ ) と上位データストロブ ( $\overline{UDS}$ ), 下位データストロブ ( $\overline{LDS}$ ) がアサートされる。

$\overline{AS}$  信号を受けると外部デバイスはアドレスをデコードして、どのデバイスが選択されているかを知る。選択されたデバイスはデータバス上にデータを送出し、また、同時にデータ送出を行ったことを示すデータ転送アクノレッジ ( $\overline{DTACK}$ ) をアサートする。

一方プロセッサは、S4 状態の終りで  $\overline{DTACK}$  がアサートされているかどうかチェックし、アサートされていればすぐに S5 状態に入る。もしこのときアサートされていない場合は、S5 の状態に入らず Sw 状態 (ウェイト状態) に入る。この Sw 状態は図 3.3 に示すように、 $\overline{DTACK}$  がアサートされるまで挿入されるので、データアクセス時間の遅い

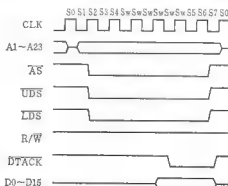


図 3.3 アクセスが遅いデバイスに対するリードサイクルタイミング

デバイスはその遅れ時間を含めて  $\overline{DTACK}$  をアサートすればよい。

プロセッサは  $\overline{DTACK}$  がアサートされてデータバス上にデータが確定していることを知ると、S6 状態の終りでデータを内部にラッチする。最後に S7 状態で、 $\overline{UDS}$ ,  $\overline{LDS}$  をネグートし、 $\overline{AS}$  をネグートしてリードサイクルを終了する。なおアドレスバスは、S7 状態が完了するまで有効である。

リードサイクルが終了すると、外部デバイスもデータ送出を停止するとともに、 $\overline{DTACK}$  をネグートする。以上、リードサイクルのオペレーションをまとめてフローチャートにすると図 3.4 のようになる。

**b. ライトサイクル** ライトサイクルはプロセッサが外部デバイスに対してデータを書き込むオペレーションである。図 3.5 はライトサイクルのタイミングチャートである。ライトサイクルにも、ワードライト、上位バイトライト、下位バイトライトのサイクルがあるが、その違いはリードサイクルの場合と全く同じである。以下ワードライトサイクルのオペレーションの説明を行う。

まず、S0 状態で、プロセッサはファンクションコードを FC0~FC1 へ出力する。このときアドレスバスはまだハイインピーダンス状態にある。S1 状態になるとアドレスバスのハイインピーダンス状態が解除され、そこからアクセスすべきデバイスのアドレスが

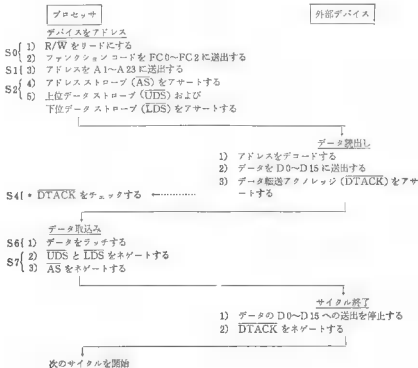


図 3.4 ワードリードサイクルフローチャート

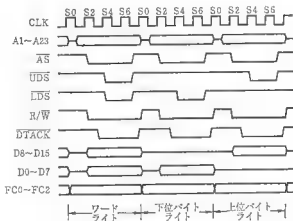


図 3.5 ライトサイクルタイミング

送出される。次の S2 状態ではアドレスストロブ ( $\overline{AS}$ ) がアサートされ、R/ $\overline{W}$  が “Low” レベルになってライトサイクルであることを示す。S3 状態になるとデータバスからデータが送出され、また、S4 状態で上位データストロブ ( $\overline{UDS}$ )、下位データストロブ ( $\overline{LDS}$ ) がアサートされる。

外部デバイス側では送出されたアドレスをデコードして、どのデバイスが選択されたかを知る。選択されたデバイスはデータバス上のデータを  $\overline{UDS}$  または  $\overline{LDS}$  を用いてメモリないしレジスタへ取り込み、プロセッサに対して  $\overline{DTACK}$  をアサートする。なお、プロセッサでは  $\overline{DTACK}$  がアサートされているかどうかをリードサイクルと同じように S4 の終りでチェックする。このとき  $\overline{DTACK}$  がアサートされていなければ Sw 状態が S5 の前に挿入されるので、書き込みに時間のかかるデバイスでは  $\overline{DTACK}$  のアサートを遅らせればよい。

プロセッサは S4 または Sw 状態で  $\overline{DTACK}$  がアサートされると、S5 の状態へ入り S6, S7 の状態へと進む。S7 状態では  $\overline{UDS}$ ,  $\overline{LDS}$  をネゲートし、さらに、 $\overline{AS}$  をネゲートする。データバス上のデータおよび R/ $\overline{W}$  のライト状態は S7 状態の間は保持され、次のサイクルの S0 状態になって始めて、データバス上からデータが除去され、R/ $\overline{W}$  がリード状態に戻る。

また、デバイス側では  $\overline{AS}$  がネゲートされたのに応じて  $\overline{DTACK}$  をネゲートする。

以上のライトサイクルのオペレーションをまとめてフローチャートにすると、図 3.6 のようになる。

**c. リードモディファイライトサイクル** リードモディファイライトサイクルは、プロセッサが外部のデバイスからデータを読み取り、それに論理演算操作を加えて変更したデータを同じアドレスのデバイスに書き込むオペレーションである。このサイクルはリードとライトオペレーションが連続して実行され、分割できないバスサイクルである。これはリードモディファイライトサイクルの重要な特徴である。リードモディファイライトサイクルはマルチプロセッサシステムにおけるプロセッサ間の交通で非常に有用な役割を果たす。すなわち、2つ以上のプロセッサ間で共有されるデバイスまたはメモリを1つのプロセッサがリードモディファイライトサイクルで使用している間は、他のプロセッサはこのデバイスまたはメモリをアクセスすることができない。したがって、プロセッサ間の交通のために必要なフラグ等に対するリード/ライトの競合を避けることができる。

68000 では TAS (Test and Set) 命令だけがリードモディファイライトサイクルを使用する。TAS 命令はバイト操作しかできないので、ワードリードモディファイライトサ



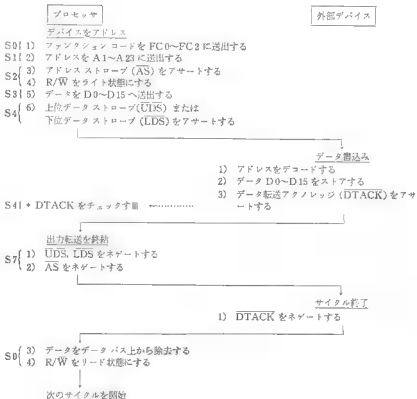


図 3.6 ワードライトサイクルフローチャート

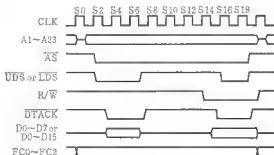


図 3.7 バイトリードモディファイライトサイクルタイミング

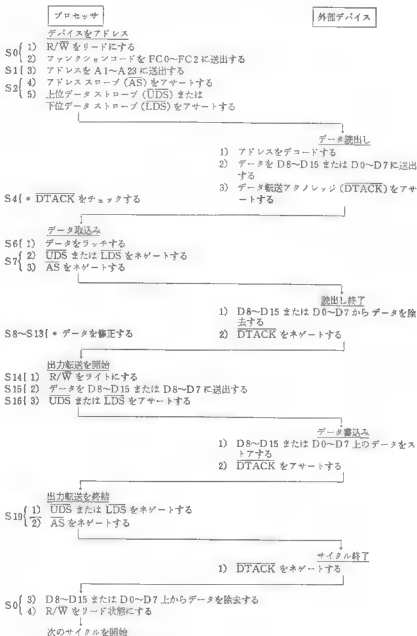


図 9.8 バイトリードモディファイライトサイクルフローチャート

イタルは存在しない。バイトリードモディファイライトサイクルのタイミングチャートを図3.7に。また、フローチャートを図3.8に示す。リードモディファイライトサイクルはリードサイクルとライトサイクルが続けて実行されるだけなので、詳細な動作の説明はここでは省略する。ただし、アドレスストロブ ( $\overline{AS}$ ) はこのサイクルの■ずっとアサートされるので注意されたい。

### 3.2.2 バスアービトレーション (Bus Arbitration)

バスアービトレーションは外部デバイスたとえば  $DMAC^{\dagger}$  がプロセッサから独立してデータバスを使用するときに必要となるオペレーションである。すなわち、バス制御能力を有するデバイス (プロセッサを含む) が2つ以上で1つのバスを共有すると、バスアクセス競合の問題が必ず生ずる。この競合を避けるためには、バス制御の支配権をどのデバイスがもつか、すなわち、どのデバイスがバスマスタになるかを決める必要がある。このオペレーションをバスアービトレーションという。

68000 ではバスアービトレーションを可能にするために、バス要求 ( $\overline{BR}$ )、バスグラント ( $\overline{BG}$ )、バスグラントアクノレッジ ( $\overline{BGACK}$ ) という3本の制御信号が用意されている。アービトレーション回路はプロセッサの外に、デジタイズまたはブライオリティエンコーダ等を用いた制御回路として構成される。この場合、バスマスタになり得る外部デバイスはいくつあってもよいが、プロセッサは外部デバイスよりもバスマスタ権は低く格付けされることになる。

図3.9はバスアービトレーションのタイミングチャートである。以下この図に従いながら、バスアービトレーションのオペレーションを説明する。

バス制御能力を有するデバイスがバスを使用したいときには、まず、プロセッサに対してバス要求 ( $\overline{BR}$ ) をアサートする。このようなデバイスが複数個あるときは、ワイヤドオアして  $\overline{BR}$  へ入力する。プロセッサのオペレーションとこの信号は全く非同期であり、プロセッサはクロック (CLK) でこの信号をサンプリングし、同期化する。したがって、プロセッサが  $\overline{BR}$  のアサートを内部的に認知するには  $\overline{BR}$  入力後約 0.5~1.5 クロックサイクルの時間が必要である。

プロセッサは  $\overline{BR}$  がアサートされたことを内部で認知すると、プロセッサが S0 状態でない場合は次のクロックサイクルで、また、S0 状態の場合は次の次のクロックサイクルでバスグラント ( $\overline{BG}$ ) をアサートする。すなわち、 $\overline{BG}$  はプロセッサの実行状態のいか

<sup>†</sup>  $DMAC$ : Direct Memory Access Controller (10章参照)



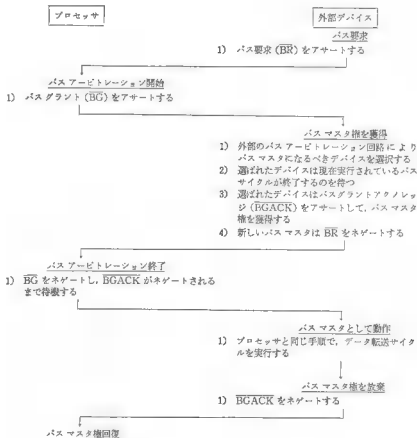


図 3.10 バスアービトレーションフローチャート

サイクルは  $\overline{\text{BGACK}}$  および  $\overline{\text{DTACK}}$  がネゲートされることによって知ることができる。

以上のようにして、 $\overline{\text{BR}}$  をアサートしたデバイスがバス マスタ権を獲得すると、そのことをプロセッサに知らせるためにバス グラント アクノレッジ ( $\text{BGACK}$ ) をアサートするとともに、 $\overline{\text{BR}}$  をネゲートする。プロセッサは  $\overline{\text{BR}}$  がネゲートされたのを受けて  $\overline{\text{BG}}$  をネゲートしバス アービトレーションを終了する。プロセッサは其の後は  $\overline{\text{BGACK}}$  がネゲートされることを待つだけである。

新しいバス マスタは、アドレス バス、データ バスを  $\overline{\text{AS}}$ 、 $\overline{\text{UDS}}$ 、 $\overline{\text{LDS}}$ 、 $\text{R/W}$  およびファンクション コードで適当に制御することにより、データ転送のバス サイクルを形成することができる。図 3.9 では新しいバス マスタが形成するバス サイクルは 1 バス サイクル

ルしか示していないが、2つ以上であってもよい。しかし、このときバスマスタは必ず  $\overline{\text{BGACK}}$  をアサートし続けておかねばならない。 $\overline{\text{BGACK}}$  は所定のバスサイクルが完全に終了してからネゲートする。これは、外部デバイスのバスマスタ権の放棄を意味する。

プロセッサは  $\overline{\text{BGACK}}$  がネゲートされたことを知ると、他の外部デバイスからのバス要求 ( $\overline{\text{BR}}$ ) が受け付けられていないかぎり、バスマスタ権を回復する。もし、ここで他のデバイスが  $\overline{\text{BR}}$  をアサートしていれば、外部のアービトレーション制御回路で再アービトレーションを行い、外部デバイスが先にバスを占有することになる。

以上、バスアービトレーションのオペレーションをまとめると、図3.10のフローチャートのようになる。

### 3.2.3 バスエラーおよびホールドオペレーション

68000のデータ転送バスオペレーションは、 $\overline{\text{DTACK}}$  信号を用いた非同期のハンドシェークデータ転送方式である。もし、何らかの原因で、たとえば、アドレスされたデバイスが存在しなかったり、電源が供給されていなかったりすると、ハンドシェークが成立し得ない。このような事態をここではバスエラーとよぶことにする。バスエラーの検出は外部回路にまかされているので、バスエラーとしてどういうものを設定するかはユーザの自由である。外部でバスエラーが検出されると、 $\overline{\text{BERR}}$  の入力信号を介してプロセッサにバスエラーの発生が知らされる。プロセッサとしてはバスエラーに対する処置は、 $\overline{\text{BERR}}$  入力ほかに  $\overline{\text{HALT}}$  信号がアサートされているかどうかで、再実行するかあるいは例外処理をするかが決められる。この項ではバスエラー時のバスオペレーションおよびホールド時のバスオペレーションについて説明する。

**a. バスエラー例外処理** バスエラー例外処理が行われるのは、 $\overline{\text{HALT}}$  信号がアサートされずに、 $\overline{\text{BERR}}$  だけがアサートされたときである。図3.11に示すようにプロセッサは、まず、 $\overline{\text{BERR}}$  がアサートされたことを S4 または Sw のクロックの立下りで検出すると、データバスおよびアドレスバスをハイインピーダンス状態にする。プロセッサは次に  $\overline{\text{BERR}}$  がネゲートされたことを知って、例外処理シーケンスを開始する。

**b. バスサイクルの再実行** プロセッサがバスサイクルを実行中に  $\overline{\text{BERR}}$  および  $\overline{\text{HALT}}$  がともにアサートされると、プロセッサはそのバスサイクルを再実行する。図3.12にバスサイクル再実行のタイミングチャートを示す。

プロセッサは、 $\overline{\text{BERR}}$  と  $\overline{\text{HALT}}$  が同時にアサートされていることを S4 または Sw のクロックの立下りで検出すると、実行中のバスサイクルを終結し、データバスおよびア

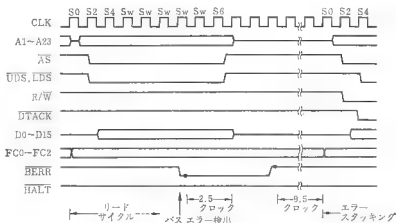


図 3.11 バスエラー タイミング

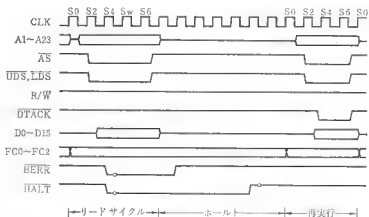


図 3.12 再実行バスサイクル タイミング

ドレスバスをハイインピーダンス状態にする。そして、プロセッサは  $\overline{\text{HALT}}$  がネゲートされるまでホールド状態になる。次に  $\overline{\text{HALT}}$  がネゲートされるとプロセッサは前と全く同じアドレス、データ、制御のもとにバスサイクルを再実行する。ここで、 $\overline{\text{HALT}}$  のネゲートは必ず  $\overline{\text{BERR}}$  をネゲートして 1 クロック後に降に行わねばならない。そうでないときは不正シーケンスとみなされ、ベクタ番号 0 の例外処理となるので注意が必要である。

プロセッサはリードモディファイライトサイクルに対してはバスサイクルの再実行を

行わない。これはリードモディファイライトサイクルは再実行しても結果が正しくなるとはかぎらないからである。リードモディファイライトサイクル実行中に再実行の要求があったときには、バスエラー例外処理となる。

c. ホールトオペレーション  $\overline{\text{BERR}}$  がネゲートのとき  $\overline{\text{HALT}}$  だけがアサートされると、プロセッサはホールト状態になる。すなわち、プロセッサは実行中のバスサイクルが終了すると、アドレスバスおよびデータバスをハイインピーダンス状態にして、 $\overline{\text{HALT}}$  がネゲートされるまで次のバスサイクルの実行を停止する。図 3.13 はホールトオペレーションを図示したものである。ここで、 $\overline{\text{HALT}}$  がアサートされても  $\overline{\text{DTACK}}$  がネゲートされたままであればバスサイクルは終了しないので、 $\overline{\text{DTACK}}$  がアサートされ

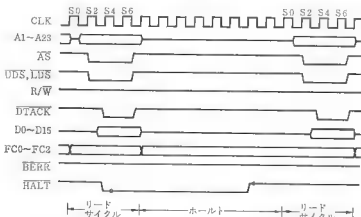
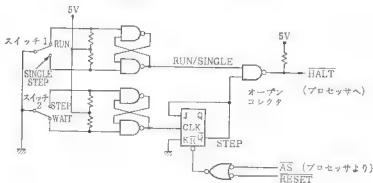


図 3.13 ホールトオペレーション



■ 3.14 簡易形シングルステップ回路



ないかぎりホールド状態にならないことに注意が必要である。

ホールドオペレーションを用いると、シングルステップでのバスサイクルの実行が可能である。図3.14はシングルステップを実現する簡易な回路である。すなわち、スイッチ1をシングルステップ側へセットすれば、バスサイクルが1つ実行されるたびに $\overline{AS}$ によってJKフリップフロップがリセットされるのでプロセッサはホールド状態になる。また、このときスイッチ2をウニートからステップ側へ1回倒すたびにJKフリップフロップはセットされるので、ホールド状態が解除されバスサイクルが1回だけ実行される。なお、この回路は、 $\overline{HALT}$ と $\overline{BERR}$ や $\overline{RES}$ の相互の影響については考慮していないので、実際に使うにあたっては注意が必要である。

プロセッサは $\overline{HALT}$ 信号を受け付けていても、バスアービトレーションは通常どおり行われる。すなわち、 $\overline{HALT}$ はバスアービトレーションには何ら影響がない。バスアービトレーションは $\overline{AS}$ 、 $\overline{LDS}$ 、 $\overline{UDS}$ 、 $R/W$ の信号をバスから除去する。つまり、ハイインピーダンス状態にする機能である。

**d. 2重バス障害 (Double Bus Faults)** バスエラー例外処理においてプロセッサはマシンの状態に関する数ワードの情報をメモリ上に退避する。もし、バスエラーがこのデータ転送サイクルで発生すれば、バスエラーが2回続けて生じたことになる。これを2重バス障害という。2重バス障害になるとプロセッサはホールドする。一度バスエラー例外が発生したとき、次の命令を実行する前に発生するバスエラー例外はすべて2重バス障害となる。

再実行バスサイクルはバスエラー例外にも、2重バス障害にもならないことに注意すべきである。プロセッサは外部回路が再実行を要求するかぎり同じバスサイクルを何度でも繰り返す。

$\overline{BERR}$ 信号はプロセッサが外部からリセット信号を受けたときのオペレーションにも影響を与える。プロセッサは $\overline{RES}$ がアサートされるとプログラムの実行を開始すべく、リセット例外処理(8.3節参照)の実行によってメモリ上の情報を読み出す。もし、このときバスエラーが生じたら、これは2重バス障害とみなされ、プロセッサはホールドする。

2重バス障害のために生じたホールド状態は、外部からのリセットでなければホールド状態を解除することはできない。

### 3.2.4 リセットオペレーション

68000のリセット端子( $\overline{RES}$ )は双方向性信号である。これにより外部回路あるいはプ

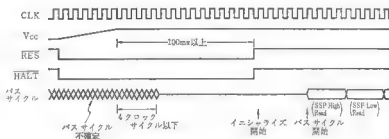


図 3.15 リセットオーバーラップ

ロセッサのいずれもがプロセッサを含む全体システムをリセットすることができる。図 3.15 はリセットオーバーラップのタイミングチャートである。プロセッサを完全にリセットするためには、 $\overline{\text{RES}}$  と  $\overline{\text{HALT}}$  が外部から同時にアサートされなければならない。このアサートしている時間は、パワーオン時で 100ms、また、通常時で 10 クロックサイクルの時間が必要である。

プロセッサはリセットを受け付けると、ベクタ番号 0 (0~3 番地) のアドレス情報を読み取りそれをスーパーバイザシステムスタック (SSP) へロードする。次にベクタ番号 1 (4~7 番地) のアドレス情報を読み取りそれをプログラムカウンタへロードする。さらにプロセッサは、ステータスレジスタの S ビットをセット、T ビットをリセットし、割込みレベルを 7 にセットする。他のレジスタはリセットオーバーラップによって影響されない。

プロセッサが RESET 命令を実行すると  $\overline{\text{RES}}$  端子は 124 クロックサイクルの間 "Low" レベルに駆動される。これは、プロセッサが外部回路をリセットする目的のものである。したがって、RESET 命令が終了すると、 $\overline{\text{RES}}$  端子につながれている外部デバイスはすべてリセットされる。なお、RESET 命令の実行はプロセッサの内部状態やレジスタには全く影響を与えない。

## 4 命令の形式とアドレス形式

68000の大きな特徴のひとつは、豊富なアドレス形式を備えていることである。この章では、まず、この豊富なアドレス形式が機械語レベルでどのようにして実現されているかについて述べ、次に各アドレス形式の機能について述べる。

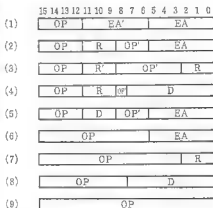
### 4.1 命令の形式

68000の命令は図4.1に示すような、4種類のワードすなわちオペレーションワード、イミディエイトオペランドワード、ソース実効アドレス拡張ワード、デスティネーション実効アドレス拡張ワードの組合せで構

成される。オペレーションワードは命令の基本部分であり、命令の機能はこれによって決る。イミディエイトオペランドワードおよびソース/デスティネーション実効アドレス拡張ワードは各々1ワードないし2ワードで構成されるが、命令



図 4.1 命令の形式



OP: オペレーションコード, EA: 実効アドレス, R: レジスタ No., D: データまたはディスプレースメント

図 4.2 オペレーションワードの形式

によっては省略される場合もある。また、イミディエイトオペランドとソース実効アドレス拡張ワードは同一命令のなかに同時には存在しない。したがって、68000の命令長は1ワードから5ワードまでとなる。

オペレーションワードは命令の基本操作を指定する部分とオペランドを指定する部分により構成される。図4.2は68000の命令のオペレーションワードの形式を示したものである。ここで、OPフィールドはオペレーションを指定する部分であり、EA、RおよびDフィールドはオペランドを指定する部分である。したがって、(1)~(5)の形式の命令は2つのオペランドをもち、(6)~(8)の形式の命令は1つのオペランドをもっている。また、(9)のようにオペランドを全くもたない命令も存在する。また、(6)~(8)の命令形式の中には、オペレーションコードによってオペランドを暗黙的に指定しているものもある。

## 4.2 アドレス形式

オペランドがどこに存在するかを指定する形式をアドレス形式といい、各命令においてオペランドが実際に存在する場所を示したものを実効アドレスという。

68000の命令では多くの場合、実効アドレスはオペレーションワードの中の実効アドレスフィールドで指定される。実効アドレスフィールドは図4.3に示すようにアドレス形式フィールドとレジスタフィールドに分けられ、これらの各フィールドがとる値によって表4.1に示すような12のアドレス形式に分けることができる。すなわち、68000の14個のアドレス形式のうち12の形式はこの実効アドレスフィールドで指定される。



図 4.3 実効アドレスフィールドの構成

図4.2のRフィールドはレジスタの番号を表しているが、これは実効アドレスフィールドのうちアドレス形式フィールドがなくなったものと考えればよい。したがって、Rフィールドで指定されるアドレス形式は表4.1の分類に含まれる。なお、図4.2における形式(8)の命令は実効アドレスとしてディスプレースメント付プログラムカウンタ相対形式が暗黙的に仮定されており、Dフィールドはそのディスプレースメントになっている。

表4.1に含まれない3つのアドレス形式は、クイックイミディエイト形式およびインプライド形式とよばれるものである。クイックイミディエイト形式とはオペレーションワードの中にイミディエイトデータをもつものをいい、■4.2の形式(4)、(5)でDフィ

表 4.1 実効アドレスフィールドによるアドレス形式

モードフィールド	レジスタフィールド	ア ド レ ス 形 式
0 0 0	Dn	データレジスタ直接形式
0 0 1	An	アドレスレジスタ直接形式
0 1 0	An	アドレスレジスタ間接形式
0 1 1	An	ポストインクリメントアドレスレジスタ間接形式
1 0 0	An	プリデクリメントアドレスレジスタ間接形式
1 0 1	An	ディスプレースメント付アドレスレジスタ間接形式
1 1 0	An	インデックス付アドレスレジスタ間接形式
1 1 1	0 0 0	短絶対アドレス形式
1 1 1	0 0 1	長絶対アドレス形式
1 1 1	0 1 0	ディスプレースメント付プログラムカウンタ相対形式
1 1 1	0 1 1	インデックス付プログラムカウンタ相対形式
1 1 1	1 0 0	イミディエイト形式

注) Dn: データレジスタ番号, An: アドレスレジスタ番号

ールドがデータである場合がこれにあたる。また、インプライド形式とはオペレーションフィールドが暗黙的にオペランドを指定するものをいう。

以下、表 4.1 のアドレス形式およびタイクイイミディエイト形式、インプライド形式について主として MOVE 命令を例にとりながら説明する。

#### 4.2.1 データレジスタ直接形式

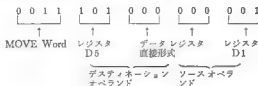
この形式ではレジスタフィールドで指定される番号  $n(=0\sim7)$  のデータレジスタ Dn がオペランドとなる。したがって、実効アドレス EA は

$$EA = Dn$$

となる。

【例】 MOVE .W D1, D5

この命令は D1 のデータを D5 へ転送する命令である。



#### 4.2.2 アドレスレジスタ直接形式

この形式ではレジスタフィールドで指定される番号  $n(=0\sim7)$  のアドレスレジスタ

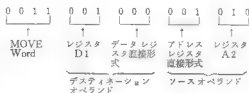
An がオペランドとなる、実効アドレス EA は

$$EA = An$$

である。

[例] MOVE.W A2, D1

この命令は A2 のデータを D1 へ転送する命令である。



#### 4.2.3 アドレスレジスタ間接形式

この形式の実効アドレスは

$$EA = (An)$$

である。すなわち、アドレスレジスタ An の内容がアクセスされるメモリのアドレスを指し示している。

[例] MOVE.W (A2), D3



この命令の動作は図 4.4 に示すとおりである。

#### 4.2.4 ポストインクリメントアドレスレジスタ間接形式

この形式は、前項のアドレスレジスタ間接形式と実効アドレスの求め方は同じであるが、命令実行の最後にアドレスレジスタがインクリメントされる点と異なる。インクリメントされる値はその命令がバイト、ワード、または、ロングワードのどの長さを実行する命令かによって、1、2 または 4 となる。すなわち、

$$EA = (An); An + N \rightarrow An \quad (N=1, 2 \text{ or } 4)$$

ただし、アドレスレジスタがスタックポインタ (A7) であるときは、バイト操作命令で

もインクリメント値は2となる。これはシステムスタックポインタをワードの境界にしておくためである。

【例】 MOVE.W (A2)+, D3



この命令の動作は図4.5に示すとおりである。

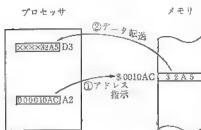


図 4.4 アドレスレジスタ間接形式の動作

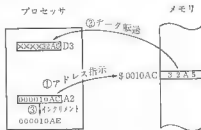


図 4.5 ポストインクリメントアドレスレジスタ間接形式の命令の動作

#### 4.2.5 プリテクリメントアドレスレジスタ間接形式

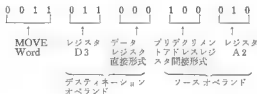
この形式ではまずアドレスレジスタの内容をデクリメントし、そのデクリメントされた値が実効アドレスを示す。すなわち、

$$An - N \rightarrow An \quad (N=1, 2 \text{ or } 4)$$

$$EA = (An)$$

となる。なお、デクリメント値  $N$  の決定方法は前項と同じく、命令が対象とするオペランドのデータ長がバイトかワードかロングワードかにより、1, 2 または 4 となる。さらに、アドレスレジスタがシステムスタックポインタであれば、バイト操作命令であってもデクリメント値は2となる。

【例】 MOVE.W -(A2), D3



この命令の動作は図 4.6 に示すとおりである。

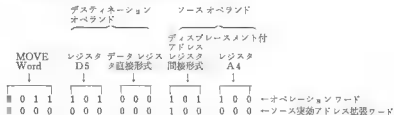
#### 4.2.6 ディスプレースメント付アドレス レジスタ間接形式

このアドレス形式は 1 ワードの実効アドレス拡張ワードが必要であり、拡張ワードはこの場合ディスプレースメントとよばれる。オペランドのアドレスはアドレスレジスタの内容と 32 ビットに符号拡張されたディスプレースメントの和になる。すなわち、実効アドレスは

$$EA = (An) + d_{16} \quad d_{16}: 16 \text{ ビットディスプレースメントの値}$$

で与えられる。

【例 1】 MOVE .W \$20 (A4), D5



この命令の動作は図 4.7 に示すとおりである。

【例 2】 MOVE .W \$20 (A4), \$10 (A3)



この例はソースオペランドもデスティネーションオペランドもディスプレースメント付アドレスレジスタ間接形式である場合を示したものである。実効アドレスの拡張はソー



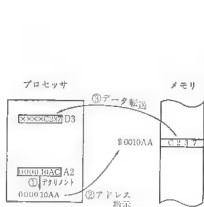


図 4.6 プリディリメントアドレスレジスタ間接形式の命令の動作

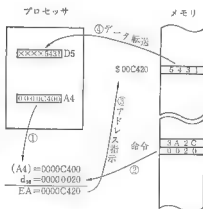


図 4.7 ディスプレースメント付アドレスレジスタ間接形式の命令の動作

ス、デスティネーション両方について前記のように各々行われる。

#### 4.2.7 インデックス付アドレスレジスタ間接形式

このアドレス形式はインデックスレジスタを指定するために1ワードの実効アドレス拡張ワードが必要である。拡張ワードは図 4.8 の形式をしている。データレジスタもアド



- D/A: 0 のときデータレジスタ, 1 のときアドレスレジスタ  
 Rn : インデックスレジスタ No.  
 W/L: 0 のときインデックスレジスタの下位ワードが符号拡張される  
 1 のときインデックスレジスタ倍長ワードがそのままとられる  
 d<sub>8</sub> : 8 ビットのディスプレースメント

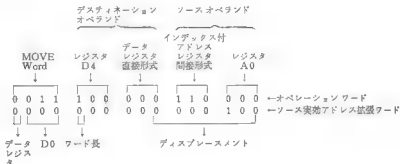
図 4.8 インデックスレジスタ指定のための拡張ワード

レスレジスタもインデックスレジスタになり得る。このアドレス形式の実効アドレスは次の式で与えられる。

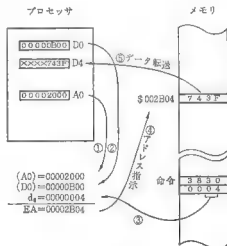
$$EA = (An) + (Ri) + d_8$$

ここで、d<sub>8</sub> は 8 ビットのディスプレースメントで 32 ビットに符号拡張される。また Ri はアドレスレジスタ A<sub>i</sub>、データレジスタ D<sub>i</sub> のいずれかである (i=0~7)。

[例] MOVE.W \$04 (A0, D0), D4



この命令の動作は図 4.9 に示すとおりである。



■ 4.9 インデックス付アドレスレジスタ間接形式の命令の動作

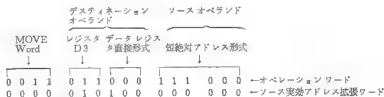
#### 4.2.8 短絶対アドレス形式

このアドレス形式では、1ワードの実効アドレス拡張ワードが必要である。実効アドレスとしてはこの拡張ワードの内容がとられる、すなわち、

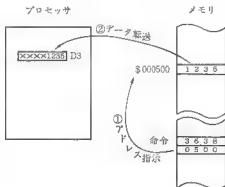
$$EA = \text{Next Word}$$

ここで、拡張ワードは16ビットであるので実効アドレスにするときには32ビットに符号拡張される。

【例】 MOVE .W \$0500, D3



この命令の動作は図 4.10 に示すとおりである。



■ 4.10 短絶対アドレス形式の命令の動作

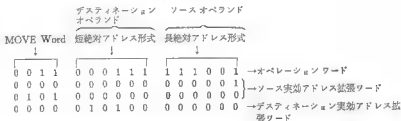
#### 4.2.9 長絶対アドレス形式

このアドレス形式は2ワードの実効アドレス拡張ワードが必要であり、実効アドレスとしてこの2ワードの内容がとられる。すなわち、

$$EA = \text{Next Two Words}$$

ここで、拡張ワードの第1ワードはアドレスの上位部分であり、第2ワードはアドレスの下位部分である。

【例】 `MOVE.W $015000, $0500`



この命令の動作は図 4.11 に示すとおりである。

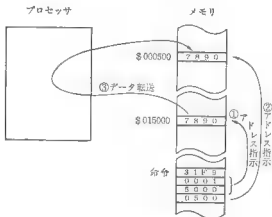


図 4.11 長絶対アドレス形式の命令の動作

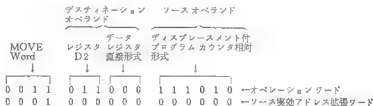
## 4.2.10 ディスプレースメント付プログラムカウンタ相対形式

このアドレス形式は 1 ワードの実効アドレス拡張ワードが必要である。実効アドレスはプログラムカウンタの内容と拡張ワードであるディスプレースメントの和である。すなわち、

$$EA = (PC) + d_{16}$$

ここで、ディスプレースメント  $d_{16}$  は 16 ビットであるが、EA 計算時には 32 ビットに符号拡張される。また、プログラムカウンタの内容は拡張ワードのアドレスを指している。

〔例〕 `MOVE .W (LABEL), D3`



この命令の動作は図 4.12 に示すとおりである。

## 4.2.11 インテックス付プログラムカウンタ相対形式

このアドレス形式は 1 ワードの実効アドレス拡張ワードが必要である。拡張ワードの形式は図 4.8 と同じであり、実効アドレスの計算式もアドレスレジスタがプログラムカウンタ

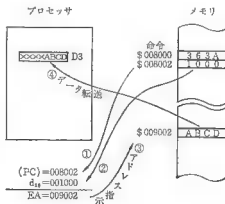


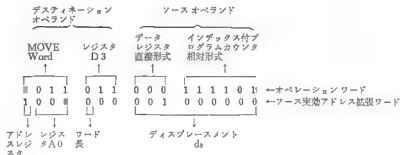
図 4.12 ディスプレースメント付プログラムカウンタ相対形式の命令の動作

ンタに変わだけである。すなわち、

$$EA = (PC) + (Ri) + d_s$$

ここで、ディスプレースメント  $d_s$  は 8 ビットであるが 32 ビットに符号拡張される。また、プログラムカウンタは拡張ワードのアドレスを指している。

[例] MOVE.W (LABEL) (A0), D3



この命令の動作は図 4.13 に示すとおりである。

#### 4.2.12 イミディエイト形式

このアドレス形式は 1 ワードまたは 2 ワードの実効アドレス拡張ワードが必要である。この拡張ワードはイミディエイトオペランドとよばれ、これが実効アドレスとなる。すなわち、

$$EA = \text{Next Word or Next Two Words}$$

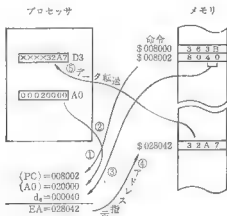


図 4.13 インデックス付プログラム  
カウンタ相対形式の命令の動作

ここで、イミディエイトオペランドは命令がバイトまたはワード操作命令のとき1ワードであり、ロングワード操作命令のとき2ワードである。

【例】 MOVE .W #\$2345, \$1000

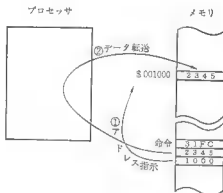


図 4.14 イミディエイト形式の命令の  
動作

この命令の動作は図4.14に示すとおりである。

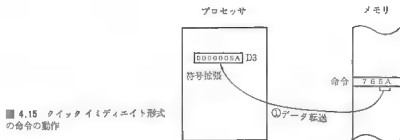
#### 4.2.13 クイックイミディエイト形式

クイックイミディエイト形式はイミディエイトデータが拡張ワードに存在するのではなく、オペレーションワードそのものに存在する。図4.2の命令の形式でいえば、(4)および(5)であり、このアドレス形式をもっている命令は3個である。すなわち、(4)の形式の命令として **MOVEQ** 命令、また、(5)の形式の命令として **ADDQ** 命令および **SUBQ** 命令があるだけである。しかも、イミディエイトデータとして取り扱うことのできるデータの範囲も、(4)の形式で-128~127、また(5)の形式で1~8と非常に小さい。しかし、このアドレスの形式は命令ワード数を減らし、また、オペランドのリードサイクルも不要なため実行時間を短くするのに大きく貢献する。

【例】 **MOVEQ ##5A, D3**



この命令の動作は図4.15に示すとおりである。



■ 4.15 クイックイミディエイト形式の命令の動作

#### 4.2.14 インブライド形式

インブライド形式とはオペレーションコードで暗黙的にオペランドが指定されるものをいう。たとえば、図4.2の(9)の形式の命令であっても、**RTS** (Return from Subroutine)、**RTE** (Return from Exception)、**RTR** (Return from Subroutine Restore CC) 命令では、オペランドとしてスタックポインタやステータスレジスタまたはコンディションコードレジスタが暗黙的に指定されている。また、(6)、(7)の形式の命令である **MOVE**

to SR (Status Register), MOVE from SR, MOVE to CCR (Condition Code Register), MOVE to USP (User Stack Pointer), MOVE from USP の各命令については、ソースまたはデスティネーションの片方のオペランドは実効アドレスフィールドで指定するが、もう片方は暗黙的に指定されている形となっている。



## 5 命令の種類

この章では68000の命令セットを機能別に分類し、それぞれについて概説する。命令セットには、以下のオペレーションを実行する機能が含まれる。

- |               |                        |
|---------------|------------------------|
| (1) データ転送命令   | (Data Movement)        |
| (2) 算術演算命令    | (Integer Arithmetic)   |
| (3) 論理演算命令    | (Logical)              |
| (4) 10進演算命令   | (Binary Coded Decimal) |
| (5) 桁移動命令     | (Shifts and Rotates)   |
| (6) ビット操作命令   | (Bit Manipulation)     |
| (7) プログラム制御命令 | (Program Control)      |
| (8) システム制御命令  | (System Control)       |

命令によってはアドレス形式やステータスレジスタの状態と無いつながりをもつものもあるので、4章あるいは6章を適宜参照されたい。

### 5.1 データ転送命令

データ転送(MOVE)命令によって、基本的なデータ転送を行うことができる。この命令セットの中には一般のデータ転送命令以外に次に示すようないくつかの特殊なデータ転送命令がある。

- |       |                           |
|-------|---------------------------|
| MOVEM | (Move Multiple Registers) |
| MOVEP | (Move Peripheral)         |
| EXG   | (Exchange Registers)      |
| LEA   | (Load Effective Address)  |

表 5.1 データ転送命令の一覧表

命 令	サイズ	操 作	特 徴
MOVE	B W L	$(EA)_s \rightarrow EA_d$	ソース、デスティネーションに独立のアドレスモードが使える
MOVEA	W L	$(EA)_s \rightarrow An$	デスティネーション = $An$ の場合 W サイズの転送は符号拡張する
MOVEM	W L	$(EA)_s \rightarrow An, Dn$ または $An, Dn \rightarrow EA$	レジスタとメモリ間のブロック転送を行う
MOVEP	W L	$(EA)_s \rightarrow Dn$ または $Dn \rightarrow EA_d$	8ビット系ペリフェラルデータの転送に便利
MOVEQ	B	$\#x \times x \rightarrow Dn$	8ビットイミディエイトデータのセット
EXG	L	$Rx \leftrightarrow Ry$	レジスタ間のデータ交換用
SWAP	L	$Dn(31:16) \leftrightarrow Dn(15:0)$	同一レジスタ内の上位ワードと下位ワードのデータ交換用
LINK	—	$An \rightarrow SP@-$ $SP \rightarrow An$ $SP+d \rightarrow SP$	サブルーチン等のデータエリアのネスタイン グ用
UNLK	—	$An \rightarrow SP$ $SP@+ \rightarrow An$	
LEA	L	$EA \rightarrow An$	実効アドレスを直接ロードする
PEA	L	$EA \rightarrow SP@-$	

S: ソース, d: デスティネーション, B: バイト, W: ワード, L: ロングワード, EA: 実効アドレス  
 ■ は間接アドレス形式を表す

PEA (Push Effective Address)

MOVEQ (Move Quick)

表 5.1 にデータ転送に関する命令の一覧表を示す。

a. 一般のデータ転送命令 MOVE 命令は 8 ビット (バイト), 16 ビット (ワード), あるいは 32 ビット (ロングワード) のデータ転送を行う。ソースおよびデスティネーションオペランドをそれぞれ独立のアドレス形式で指定できる点が、この命令の大きな特徴になっている。ソースアドレスについてはすべてのアドレス形式が可能であり、デスティネーションアドレスについてはプログラムカウンタ修飾およびイミディエイトデータ形式を除くすべてのアドレス形式が使える。ただし、デスティネーションオペランドにアドレスレジスタを指定した場合、68000 はこれを MOVEA 命令と見なす。

68000 は周辺 I/O 装置のデータをメモリのデータと全く区別せずに扱う (メモリマップド I/O) ので特別な I/O 命令をもっていない。周辺 I/O 装置とのデータ転送を行う場合にも、周辺装置に固有のアドレスを割り当て、このアドレスを指定して MOVE 命令を実行する。転送するデータの長さやアドレス形式についてもメモリデータの場合と全く

同一である。

**b. その他のデータ転送命令** データ転送命令の中には a. で述べた一般的な命令のほかにブロック転送を始めとする使用目的に応じた各種の命令が準備されている。以下、これらの命令の特徴を説明する。詳細は12章で説明する。

ブロック転送命令として **MOVEM** (Move Multiple Registers) 命令がある。この命令は内部レジスタ群とメモリとの間でのブロック転送を行う。この命令では転送したいレジスタを命令の第2ワードで指定でき、これによってデータレジスタ8個、アドレスレジスタ8個のうちの任意のレジスタのブロック転送が行える。また転送語長もバイト、ワード、ロングワードを指定できる。

**MOVEP** (Move Peripheral) 命令は特に8ビットのデータを送受する周辺LSIと68000とのデータ転送に適した命令である。データレジスタの2バイトまたは4バイトのデータと、実効アドレスで指定されたメモリの連続する2ないし4ワードの上位バイトまたは下位バイトとの間の転送を行う。指定されたアドレスが偶数か奇数かによって上位バイトか下位バイトかが決る。この命令を使用すると16ビットバスに接続された8ビット系の周辺LSIとのデータ転送が容易になる。

**MOVEQ** (Move Quick) 命令は、オペレーションワードの8ビットフィールドに書かれているイミディエイトデータをロングワードに符号拡張してデータレジスタに転送する命令である。この命令は1ワード長命令で4クロックで動作し、**MOVE** 命令に比べて高速処理される。

**EXG** (Exchange Registers) 命令は2つのレジスタ間の内容を交換する命令である。データ交換は32ビット(すなわちロングワード)単位で行い、データレジスタ間、アドレスレジスタ間およびデータレジスタとアドレスレジスタ相互で行える。さらに同一レジスタの上位ワードと下位ワードの交換には **SWAP** (Swap Register Halves) が準備されている。

実効アドレスのロード命令として **LEA** (Load Effective Address), **PEA** (Push Effective Address) 命令がある。**LEA** 命令は指定されたアドレス形式で実効アドレスを計算し、これをアドレスレジスタにロードする。また **PEA** 命令は実効アドレスをシステムスタックに格納する。

サブルーチンを多重にネスティングする場合の各サブルーチンで使用するスタックエリアの確保、解放を行う命令として **LINK** (Link and Allocate) 命令と **UNLK** (Unlink) 命令がある。**LINK** 命令はアドレスレジスタ  $A_n$  の内容をシステムスタックに格納し、

アドレスを示すシステムスタックポインタの内容をアドレスレジスタ  $A_n$  にロードする。次にサブルーチンの実行に必要なエリア分だけスタックポインタを移動する。UNLK 命令は LINK 命令で確保したスタック領域を解除して 1 段浅いネスティング状態に戻る。LINK, UNLK 命令はサブルーチンの実行に必要なデータエリアを確保しながらネスティングするので、リネントラントなサブルーチンを構成できる。LINK, UNLK 命令の詳細は 15 章で説明する。

## 5.2 算術演算命令

算術命令の中には、加算、減算、乗算、除算のほかには比較、テスト、クリア命令がある。表 5.2 に算術演算の一覧を示す。以下、機能の概要を示す。詳細は 13 章で説明する。

表 5.2 算術演算命令表

命 令	サイズ	操 作	特 長
ADD/SUB	W L	$D_n \pm (EA) \rightarrow D_n$ $(EA) \pm D_n \rightarrow EA$	ワード指定の場合、ソースオペランドをロングワードに符号拡張して演算する B, W, L の任意サイズのイミディエイトデータが扱える 1~8 ビットイミディエイトデータの処理を行う
ADDA/SUBA	W L	$A_n + (EA) \rightarrow A_n$	
ADDI/SUBI	W L	$(EA) \pm \# \times \times \times \rightarrow EA$	
ADDQ/SUBQ	B W L	$(EA) \pm \# \times \times \times \rightarrow EA$	
ADDOX/SUBX	B W L	$D_x \pm D_y \pm X \rightarrow D_x$	
	W L	$A_x @ - \pm A_y @ - \pm X \rightarrow A_x @ -$	
MULS/MULU	$W * W \rightarrow L$	$D_n * (EA) \rightarrow D_n$	
DIVS/DIVU	$L \div W \rightarrow L$	$D_n / (EA) \rightarrow D_n$	
EXT	B $\rightarrow$ W W $\rightarrow$ L	$D_n[7] \rightarrow D_n[8-15]$ $D_n[15] \rightarrow D_n[16-31]$	
NEG	B W L	$0 - (EA) \rightarrow EA$	
NEGX	B W L	$0 - (EA) - X \rightarrow EA$	
CMP	B W L	$D_n - (EA)$	複数プロセッサの同期に有効
CMPA	W L	$A_n - (EA)$	
CMPI	B W L	$(EA) - \# \times \times \times$	
CMPM	B W L	$A_x @ - A_y @ -$	
TST	B W L	$(EA) - 0$	
TAS	B	$(EA) - 0, 1 \rightarrow EA[7]$	

a. 加算・減算命令 ADD (Add Binary) 命令および SUB (Subtract Binary) 命令はデータオペレーションとアドレスオペレーションの両方に使用できる。データオペレ

ーションはデスティネーション オペランドにソース オペランドを加算あるいは減算し、その結果をデスティネーションのデータ レジスタまたはメモリに格納する。データ オペレーションではすべてのオペランドサイズが使用できる。アドレス オペレーションはデスティネーション オペランドとしてアドレス レジスタを指定する場合であり、ADD (Add Address), SUB (Subtract Address) 命令によって実行される。アドレス オペレーションのオペランドサイズは16または32ビットに限定される。ワード指定の場合にはソースオペランドをワードからロングワードに符号拡張して32ビットデータとして演算される。

イミディエイトデータの加算・減算命令として ADDI (Add Immediate), ADDQ (Add Quick), SUBI (Subtract Immediate), SUBQ (Subtract Quick) 命令がある。ADDI, SUBI 命令はデスティネーション オペランドとイミディエイトデータを加算あるいは減算を行ってデスティネーション オペランドに結果を格納する。データのサイズはバイト、ワード、ロングワードのすべてが指定できる。ADDQ, SUBQ 命令は1~8のイミディエイトデータとデスティネーション オペランド間の演算を行う。この命令は ADDI, SUBI に比べて実行タックルサイクル数およびプログラムバイト数が少ないという特徴がある。SUB 命令の変形として0からデスティネーション オペランドを減算する NEG (Negate) 命令がある。

**b. 乗算・除算命令** 乗算命令 MULS (Multiply Signed), MULU (Multiply Unsigned) 命令はワード長オペランドの乗算を行ってロングワード長の積を得る。オペランドがレジスタの場合は下位16ビットのみが取り出され上位16ビットは無視される。除算命令 DIVS (Divide Signed), DIVU (Divide Unsigned) はロングワード長のデスティネーション オペランドをワード長のソースオペランドで除算し、ワード長の商と余りを得る。商はデスティネーションの下位ワードに、また余りは上位ワードに格納される。除数が0の場合はトラップが発生し、例外処理を開始する。また命令完了前にオーバフローが生じるとVフラグがセットされ命令は実行されない。この場合デスティネーション オペランドは元の値が保持される。乗算・除算共符号付、および符号なしオペランドの演算ができる。符号付の場合は MULS, DIVS 命令を、また符号なしの場合は MULU, DIVU 命令を使用する。

**c. 比較命令** 比較命令には CMP (Compare), アドレスレジスタの比較を行う CMPA (Compare Address), イミディエイトデータとの比較を行う CMPI (Compare Immediate) メモリデータ間の比較を行う CMPM (Compare Memory) および0との比較を行う TST

(Test) がある。テスト命令の変形として、**TAS** (Test and Set Operand) 命令がある。この命令はデスティネーションオペランドのテストを行うと同時に、そのオペランドの最上位ビットを“1”にセットする。テストとセットがひとつのバスサイクル中で行われるので、複数のプロセッサの同期等に有効である。

比較命令はデスティネーションオペランドからソースオペランドを減算し、その結果に従ってコンディションコードをセットする。このときオペランド自身は変化しない。

**d. 拡張命令セット** 拡張命令セットを使用することによって、倍精度演算および長さの異なるオペランド間の算術演算を実行することができる。これらの命令としては **AD DX** (Add Multi-precision), **SUBX** (Subtract Multi-precision), **EXT** (Sign Extend), **NEGX** (Negate Multi-precision) がある。この命令はデスティネーションオペランドとソースオペランドとコンディションコードの中の X フラグの間の演算を行う。たとえば 64 ビット長のデータの加算を行う場合、下位 32 ビットの加算は **ADD** 命令で行い、上位 32 ビットの加算は **ADDX** 命令で行う。**ADDX** 命令を用いることで下位からの桁上りに対して正しく 64 ビットデータの演算が行える。

**e. クリア命令** **CLR** (Clear Operand) 命令はオペランドに 0 をセットする命令である。

### 5.3 論理演算命令

論理演算命令として **AND** (論理積), **OR** (論理和), **EOR** (排他的論理和), **NOT** (否定) 命令がある。これらはすべてのオペランドサイズに対して使用できる。また、類似のイミディエイト命令セット (**ANDI**, **ORI**, **EORI** 命令) も、すべてのオペランドサイズに対して実行できる。表 5.3 に論理演算命令の一覧表を示す。命令の詳細説明は 13 章で行う。

### 5.4 2 進化 10 進数演算命令

2 進化 10 進数 (Binary Coded Decimal) の演算命令に **ABCD** (Add Digits), **SBCD** (Subtract Digits), **NBCD** (Negate Digits) がある。これらの命令は 2 進化 10 進形式で記憶されているデータ間の加算減算処理を行い、長いデータ間の処理が容易にできる。表 5.4 に 2 進化 10 進数の演算命令表を示す。各命令の詳細は 13 章で述べる。

表 5.3 論理演算命令表

命 令	サイズ	操 作
AND	B W L	$D_n \wedge (EA) \rightarrow D_n$ $(EA) \wedge D_n \rightarrow EA$
ANDI	B W L	$(EA) \wedge \# \times \times \times \rightarrow EA$
OR	B W L	$D_n \vee (EA) \rightarrow D_n$ $(EA) \vee D_n \rightarrow EA$
ORI	B W L	$(EA) \vee \# \times \times \times \rightarrow EA$
EOR	B W L	$(EA) \oplus D_n \rightarrow EA$
EORI	B W L	$(EA) \oplus \# \times \times \times \rightarrow EA$
NOT	B W L	$\neg(EA) \rightarrow EA$

表 5.4 2進化10進数演算命令表

命 令	サイズ	操 作
ABCD	B	$D_{x10} + D_{y10} + X \rightarrow D_x$ $Ax@_{-10} + Ay@_{-10} + X \rightarrow Ax@_{-}$
SBCD	B	$D_{x10} - D_{y10} - X \rightarrow D_x$ $Ax@_{-10} - Ay@_{-10} - X \rightarrow Ax@_{-}$
NBCD	B	$0 - (EA) - X \rightarrow EA$

† ファイックスの10は、データ ( $D_x, D_y$ ) およびアドレス ( $Ax, Ay$ ) が2進化10進形式であることを示している。

## 5.5 桁移動操作命令

**シフトおよびローテート命令** 算術シフト命令 **ASL, ASR** (Arithmetic Shift Left/Right), 論理シフト命令 **LSL, LSR** (Logical Shift Left/Right) によってシフト動作が行える。ローテート命令には拡張フラグを含まない **ROL, ROR** (Rotate without X Left/Right) および拡張フラグを含む **ROXL, ROXR** (Rotate through X Left/Right) がある。すべてのシフト操作とローテート操作はレジスタとメモリのどちらにおいても実行できる。レジスタのシフトとローテートではオペランドはすべてのサイズをとることができる。シフトあるいはローテートするビット数の指定法として2種類が準備されている。命令の中で直接カウント数を指定する場合は1~8ビットの範囲となる。データレジスタで間接的に指定する場合は0~63ビットの範囲が指定できる。表5.5にシフトおよびローテート命令の一覧を示す。各命令の詳細は14章で説明される。

表 5.5 算術・論理シフトおよびローテート命令表

命 令	サイズ	操 作
ASL	B W L	
ASR	B W L	
LSL	B W L	
LSR	B W L	
ROL	B W L	
ROR	B W L	
ROXL	B W L	
ROXR	B W L	

## 5.6 ビット操作命令

ビット操作命令として **BTST** (Bit Test), **BSET** (Bit Test and Set), **BCLR** (Bit Test and Clear), **BCHG** (Bit Test and Change) 命令がある。これらの命令はデスティネーションオペランドの任意の1ビットに対してビット操作を行うもので、メモリ上のデータについては8ビット、データレジスタについては32ビットのデータのうちの1ビットが指定できる。ビット位置の指定の方法にはデータレジスタで行う方法と命令の第2ワード目で指定する方法がある。**BTST** 命令は指定ビットの状態をテストし、その結果をステータスレジスタのZビットに反映する。**BSET**, **BCLR**, **BCHG** 命令はテスト操作に引き続いて、テストビットに対してそれぞれ1, 0のセットあるいは反転操作を行う。表5.6にビット操作命令の一覧を示す。各命令の詳細は14章で述べる。



表 5.8 ビット操作命令表

命 令	サイズ	操 作
BTST	B L	~bit of (EA)→Z
BSET	B L	~bit of (EA)→Z 1→bit of (EA)
BCLR	B L	~bit of (EA)→Z 0→bit of (EA)
BCHG	B L	~bit of (EA)→Z ~bit of (EA)→bit of EA

・Z←ステータスレジスタのビット 2 (Zero)

・テストするビット位置はデータレジスタで指定する方法とリテラル値で指定する方法がある。

・~はビット反転を表す (1=0, 0=1)。

## 5.7 プログラム制御命令

**a. 条件付ブランチ命令** コンディションモードの状態を判定して、その結果によってプログラムカウンタ (PC) を操作する命令として Bcc (Branch Conditionally), DBcc (Decrement Counter and Branch until Condition True or Count=-1) 命令がある。状態判定はコンディションコードの4つのフラグに対して16種類 (Bcc に対しては14種類) が用意されている。ブランチアドレスはプログラムカウンタからの相対値として与えられ、 $-2^{15} \sim 2^{15}-1$  バイト離れた位置までブランチできる。DBcc 命令はデータレジスタ Dn をカウンタとして使用し、命令実行毎に Dn の内容をデクリメントし Dn=0 になるまで分岐アドレスにブランチし、Dn=-1 になると次の命令が実行される。Scc (Set Conditionally) はコンディションコードの状態を判定し、条件が真 (True) の場合、実効アドレスで指定されたバイトデータのすべてのビットに "1" をセットし、条件が偽 (False) の場合は "0" をセットする。条件を表 5.7 に示す。演算が符号付であるかどうかにより、算術的に同じ意味でも条件が違ふ。表 5.8 の条件式とコンディション (条件) コードの対応をみて、目的に合った条件式を使用すればよい。

**b. 無条件ブランチまたはジャンプ命令** 表 5.9 にプログラム操作命令の一覧を示す。

ブランチ命令として BRA (Branch Always), BSR (Branch to Subroutine) 命令があり、ジャンプ命令として JMP (Jump), JSR (Jump to Subroutine) 命令がある。BRA 命令の分岐アドレスはプログラムカウンタから相対アドレスとして 8 または 16 ビットディ

表 5.7 条件分枝の判定条件

指 定	判 定 条 件	適用可能性		
		Bcc	DBcc	Scc
CC	キャリイ クリア	○	○	○
CS	キャリイ セット	○	○	○
EQ	等しい	○	○	○
F	真か	×	○	○
GE	>=	○	○	○
GT	>	○	○	○
HI	ハイ	○	○	○
LE	<=	○	○	○
LS	ローあるいは同じ	○	○	○
LT	<	○	○	○
MI	マイナス	○	○	○
NE	ヤ	○	○	○
PL	プラス	○	○	○
T	常に真	×	○	○
VC	オーバフローなし	○	○	○
VS	オーバフローあり	○	○	○

表 5.8 符号条件コードの対応

条件式	符号なし	符号あり
$X < Y$	HI	GT
$X \leq Y$	CC	GE
$X \geq Y$	LS	LE
$X > Y$	CS	LT

表 5.9 プログラム操作命令表

種 類	命 令	操 作
条件付操作	Bcc	条件ブランチ (8または16ビットディスプレースメント)
	DBcc	条件テスト, $D_n - 1 \rightarrow D_n$ , If $D_n = -1$ then ブランチ
	Scc	条件=1: 1→EA 条件=0: 0→EA (16ビットディスプレースメント)
無条件分枝	BRA	常にブランチ (8または16ビットディスプレースメント)
	BSR	サブルーチンブランチ ( )
	JMP	ジャンプ
	JSR	サブルーチンジャンプ
リ タ ーン	RTS	サブルーチンからのリターン
	RTR	サブルーチンからのリターンおよび CC 回復

スプレースメントで指定され、JMP 命令の分岐アドレスは種々提供されている。これについては15章で説明する。

JSR, RTS (Return from Subroutine) 命令はサブルーチンのネスティングを操作する。メインプログラムに書かれた JSR 命令を実行するとプログラムカウンタをスタックへ退避しサブルーチンにジャンプする。サブルーチンでは最後に書かれた RTS 命令を実行するとスタックの先頭のデータをプログラムカウンタに回復し、メインプログラムに戻る。

RTR (Return from Subroutine Restore CC) 命令はスタックからプログラムカウンタとコンディションコードを回復する命令である。

各命令の詳細は15章で説明される。

## 5.8 システム制御命令

表5.10にシステム制御に関する命令をまとめて示す。これらはトラップ命令、ステータスレジスタ操作命令および特権命令に分類される。トラップ命令、ステータスレジスタ操作命令は16章、特権命令は17章で詳しく説明する。本節では、以下に各命令の概要を述べる。

表 5.10 システム制御命令表

種 類	命 令	操 作
トラップ発生命令	TRAP TRAPV CHK	PC→SSP@-, SR→SSP@- (ベクトル)→PC If V=1 then TRAP If Dn<0 or Dn>(〈EA〉) then TRAP
ステータスレジスタ操作	MOVE EA to CCR MOVE SR to EA ANDI to CCR EORI to CCR ORI to CCR	(EA)→CCR (SR)→EA (CCR)∧#××××→CCR (CCR)⊕#××××→CCR (CCR)∨#××××→CCR
特 権 命 令	MOVE EA to SR ORI to SR ANDI to SR EORI to SR MOVE to/from USP  RTE RESET STOP	(EA)→SR (SR)∨#××××→SR (SR)∧#××××→SR (SR)⊕#××××→SR (USP)→An (An)→USP SP@+SR; SP@→PC 外部デバイスをリセット #××××→SR, ストップ

**a. トラップ発生命令**    トラップ発生命令はソフトウェア割込みを発生する命令で TRAP (Trap), TRAPV (Trap if Overflow Set), CHK (Check Register against Bounds) 命令がある。TRAP 命令を実行するとプログラムカウンタとステータスレジスタをスタックへ退避して、命令で指定したベクトル番号に対応するアドレスがセットされる。TRAPV 命令を実行するとコンディションコードの V フラグがテストされ、V フラグが“1”の場合は例外処理を開始する。TRAPV 例外処理ベクトルはシステムで発生されるので命令で指定しなくてよい。V フラグが“0”の場合は何もせずに直ちに次の命令に移る。CHK 命令はデータレジスタの下位 16 ビットについて判定し、0 より小さいかまたはソースオペランドで指定した上限値より大きい場合は例外処理を開始する。

**b. ステータスレジスタ操作命令**    特権命令にならないステータスレジスタ操作命令に MOVE EA to CCR (Condition Code Register), MOVE SR (Status Register) to EA, ANDI to CCR, EORI to CCR, ORI to CCR 命令がある。

**c. 特権命令**    特権命令はスーパーバイザ状態でのみ実行可能な命令で、システム制御命令のうち特にシステムの安全性に影響の大きな命令である。

ステータスレジスタを操作する特権命令として MOVE EA to SR (Move to Status Register), ORI to SR, AND to SR, EOR to SR 命令がある。いずれもステータスレジスタ 16 ビットの操作命令である。なおステータスレジスタの下位 8 ビットすなわちコンディションコードレジスタを指定した場合は特権命令にならない。

MOVE USP (Move User Stack Pointer) 命令はアドレスレジスタとユーザシステムスタックポインタ USP の間のデータの転送を行う。スーパーバイザモードでユーザシステムスタックを使う場合に必要になる。

RTE (Return from Exception) 命令は例外処理からのリターン命令で例外処理プログラムの最後に書かれる。この命令を実行するとスーパーバイザシステムスタックからステータスレジスタとプログラムカウンタの内容が回復され、例外処理発生の前の状態に戻る。

RESET 命令の実行によってリセット信号が発生し、リセットラインにつながるすべての外部装置をリセットする。また STOP (Load SR/Stop) 命令では 16 ビットのイミディエイトデータをステータスレジスタに転送し、プログラムカウンタが次のアドレスを示した状態でプロセッサは命令のフェッチおよび実行を停止する。トレース、割込み、リセットの例外処理が生じた場合に命令の実行を再開する。STOP 命令を実行する際、ステータスレジスタの T ビット (ビット 15) が“1”の場合は直ちにトレース例外処理に移る。

以上説明した命令のほかに、いずれの分類にも入らない命令として NOP (No Operation) がある。この命令の命令長は1バイトであり何もせずに次の命令に移るもので、実行サイクル数やプログラムワード数を調整する場合に有効である。

またオペレーションワードの上位4ビットが“1010”または“1111”の命令は、未実装命令とよばれ、いわゆるエミュレーション用のものである。16章で詳しく説明する。

# 6

## フラグと算術論理

この章では 68000 のフラグの種類と機能の説明を行う。図 6.1 に示すように 68000 には次の 5 個のフラグがある。

- (1) N……ネガティブフラグ (Negative)
- (2) Z……ゼロフラグ (Zero)
- (3) V……オーバーフローフラグ (Overflow)
- (4) C……キャリイフラグ (Carry)
- (5) X……エクステンドフラグ (Extend)

これらのフラグビットはまとめてコンディションコードとよばれ、コンディションコードレジスタ (CCR) に格納される。CCR はステータスレジスタの下位バイトに位置している (2 章参照)。CCR へはスーパーバイザ状態でもユーザ状態でもアクセスできる。

各フラグは算術論理演算命令、シフト命

令、ローチャート命令およびビット操作命令などの実行結果を反映する。68000 では演算のサイズとして 1, 8, 16, 32 ビットがあるが、フラグの設定は、指定されたオペランドサイズ内の実行結果を反映する。

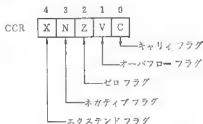


図 6.1 68000 のコンディションコードレジスタ (CCR) とフラグ構成

### 6.1 命令実行とフラグの設定

68000 が命令を実行した場合、その実行結果によって各フラグは次のいずれかの状態と

表 6.1 フラグ機能に関する記号の定義

記 号	意 味
$S_m$	ソースオペランドの最上位ビット
$D_m$	デスティネーションオペランド最上位ビット
$R_m$	実行結果の最上位ビット
$R_d$	実行結果の最下位ビット
$D_n$	デスティネーションオペランドの第nビット
$r$	右移動命令における右移動の数

なる。

- (1) 0に設定
- (2) 1に設定
- (3) 未定義 (0, 1のいずれとなるか保証されない)
- (4) 命令実行前の値を保持

各フラグの設定条件の説明に必要な記号を表 6.1 に定義しておく。

## 6.2 各フラグの説明

表 6.2 は、フラグに影響を与える命令とフラグの設定条件に関するものである。表 6.2 に現れない命令を実行してもフラグの値は変化しない。特に、アドレスレジスタを使う ADDA, SUBA, MOVEA 命令はフラグに影響しないことに注意を要する。以下、各フラグについて説明する。

### 6.2.1 N フ ラ グ

N (Negative) フラグは、命令の実行結果が負かどうかを示すフラグである。このフラグは、実行結果が負のとき 1、正または 0 のとき 0 が設定される。一般的には、実行結果が負とは最上位ビット  $R_m$  が 1 であることに等しいため、次式が成り立つ。

$$N = R_m \quad (6-1)$$

表 6.2 でも示すように大部分の命令では式 (6-1) が成立するが、DIVS, DIVU 命令、2 進化 10 進数演算命令、CHK 命令および CCR を操作する命令では式 (6-1) は成り立たない。

DIVS, DIVU 命令では、商 (結果の下位バイト) が負のとき N フラグは 1 に設定される。商のオーバフローが発生した場合には N フラグは未定義である。除数が 0 の場合 N フラ

表 6.2 フラグに影響を与える命令とフラグの設定条件

命令の種類	命 令	X	N	Z	V	C	特殊な条件
算 術 算 命 令	ADD, ADDI, ADDQ	*	*	*	!	!	$V = S_m \cdot D_m \cdot \overline{R_m} + \overline{S_m} \cdot D_m \cdot R_m$ <sup>†</sup> $C = S_m \cdot D_m + D_m \cdot \overline{R_m} + \overline{S_m} \cdot R_m$
	ADDX	*	*	!	!	!	$\left. \begin{matrix} V \\ C \end{matrix} \right\} \text{同上}$ $Z = Z \cdot \overline{R_m} \cdots \overline{R_0}$
	CLR	—	0	1	0	0	
	CMP, CMPA, CMPI, CPM	—	*	*	!	!	$V = S_m \cdot \overline{D_m} \cdot R_m + \overline{S_m} \cdot D_m \cdot \overline{R_m}$ $C = S_m \cdot \overline{D_m} + \overline{D_m} \cdot R_m + S_m \cdot R_m$
	DIVS, DIVU	—	*	*†	!	0	V=0の場合、エラー
	EXT	—	*	*	0	0	
	MULS, MULU	—	*	*	0	0	
	NEG	*	*	*	!	!	$V = D_m \cdot R_m$ $C = D_m + R_m$
	NEGX	*	*	!	!	!	$\left. \begin{matrix} V \\ C \end{matrix} \right\} \text{同上}$ $Z = Z \cdot \overline{R_m} \cdots \overline{R_0}$
	SUB, SUBI, SUBQ	*	*	*	!	!	$V = S_m \cdot \overline{D_m} \cdot R_m + \overline{S_m} \cdot D_m \cdot \overline{R_m}$ $C = S_m \cdot \overline{D_m} + \overline{D_m} \cdot R_m + S_m \cdot R_m$
	SUBX	*	*	!	!	!	$\left. \begin{matrix} V \\ C \end{matrix} \right\} \text{同上}$ $Z = Z \cdot \overline{R_m} \cdots \overline{R_0}$
	TAS, TST	—	*	*	0	0	
論 理 算 命 令	AND, AND, EOR, EORI, NOT, OR, ORI	—	*	*	0	0	
2進化10進 数演算命令	ABCD	*	U	!	U	!	C=10進数のキャリー $Z = Z \cdot \overline{R_m} \cdots \overline{R_0}$
	NBCD, SBCD	*	U	!	U	!	C=10進数のボロー $Z = Z \cdot \overline{R_m} \cdots \overline{R_0}$
桁移動命令	ASL, ASR } LSL, LSR } ROL, ROR } ROXL, ROXR(r=0)	—	*	*	0	0	
	ASL	*	*	*	!	!	C=X $V = D_m \cdot (\overline{D_{m-1}} + \cdots + \overline{D_{m-r}}) + \overline{D_m} \cdot (D_{m-1} + \cdots + D_{m-r})$ $C = D_{m-r+1}$ $C = D_{m-r+1}$
	LSL, ROL, ROXL, ASR,	*	*	*	0	!	C=D <sub>r-1</sub>
		*	*	*	0	!	
		*	*	*	0	!	
		*	*	*	0	!	



命令の種類	命 令	X	N	Z	V	C	特殊な条件
	LSR, ROR, ROXR	*	*	*	0	!	
ビット操作 命令	BCHG, BCLR, SET, BTST	—	—	!	—	—	$Z = \overline{D_n}$
その他の 命令	MOVE, MOVEQ, SWAP CHK	—	*	*	0	0	
		—	* <sup>1)</sup>	U	U	U	
	MOVE to CCR MOVE to SR RTE RTR STOP			! ! ! ! !			ソースオペランドの値 SSP の値 USP の値 イミディエイト値

—: 命令実行前の値を保持, U: 未定義, !: 特殊な条件を参照, \*: 一般の場合  
(X=C, N=結果の最上位ビット  $R_m$ , Z=演算結果が0のとき  $\overline{R_m} \cdots \overline{R_0}$ )

<sup>1)</sup> 記号の意味は表 6.1 参照

<sup>2)</sup> 商がオーバーフローの場合は未定義, 除数が0の場合は N=0, Z=1 に設定

<sup>3)</sup> 0<データレジスタ<[ea] のとき未定義

グには0が設定され、プロセッサは例外処理を開始する。

2 進化10進数演算命令では N フラグは意味をもたないので未定義である。

CHK 命令ではデータレジスタ  $D_n$  の内容により、

(1)  $D_n < 0$  のとき  $N=1$

(2)  $D_n > (EA)$  のとき  $N=0$

と N フラグが設定されトラップが生じプロセッサは例外処理を開始する。しかしデータレジスタ  $D_n$  が (1), (2) 以外有的时候には N フラグは未定義である。

MOVE to CCR/SR 命令および STOP 命令ではオペランドの対応するビットが各フラグに設定される。RTE, RTR 命令ではそれぞれシステムスタックの対応するビットが各フラグに設定される。

### 6.2.2 Z フラグ

Z (Zero) フラグは命令の実行結果がゼロかどうかを示すフラグであり、結果がゼロのとき1、そうでないとき0が設定される。一般的には、実行結果がゼロということは結果の各ビット  $R_i$  ( $i=0 \sim m$ ) が "0" ということであるから次式が成り立つ。

$$Z = \overline{R_m} \cdots \overline{R_0} \quad (6-2)$$

拡張演算命令 (ADDX, NEGX, SUBX 命令) および 2 進化10進数演算命令 (ABCD, NBCD,

SBCD 命令)では、実行が完了した桁の演算に加え下位桁の状態も反映させなければならない。すなわち、拡張語全体としてあるいは2進化10進数全桁としてゼロかどうか判断する必要がある。したがって、

$$Z = Z \cdot \overline{R_m} \cdot \dots \cdot \overline{R_0} \quad (6-3)$$

となる。ここで右辺のZは演算前すなわち下位桁の演算結果のZフラグである。2進化10進数演算命令の欄下位桁の実行に先立って、ZフラグとXフラグはクリアしておかなければならない。

大部分の命令では式(6-2)または(6-3)が成り立つが、DIVS, DIVU 命令、ビット操作命令、CHK 命令ならびにCCRを操作する命令は、これらにあてはまらない。

DIVS, DIVU 命令においては、商がゼロのときZフラグは1に設定される。商のオーバーフローが生じた場合にはZフラグの値は未定義である。除数が0のときはZフラグに1が設定されプロセッサは例外処理を開始する。

ビット操作命令(BCHG, BCLR, BSET, BTST 命令)では比較結果がZフラグに反映された後に、ビット反転、ビットクリア、ビットセットが行われる。このため命令実行後のビットの値とZフラグの値は必ずしも一致しない。

CHK 命令ではZフラグは未定義である。

### 6.2.3 V フ ラ グ

V (Overflow) フラグは命令の算術演算においてオーバーフローが生じたかどうかを示すフラグである。Vフラグには、オーバーフローが生じたとき1が設定され、生じなかったとき0が設定される。ここで、オーバーフローとは、演算結果が命令で指定したオペランドサイズ内で表現できない場合、すなわち符号ビットが破壊されてしまった場合をいう。

加算命令(ADD, ADDI, ADDQ, ADDX 命令)においてオーバーフローが生じるのは図6.2で示すように2とおりの場合がある。

(1) 負の整数(デスティネーションオペランドの最上位ビット $S_m=1$ )と負の整数(ソースオペランドの最上位ビット $D_m=1$ )を加算した結果、符号ビット(実行結果の最上位ビット $R_m$ )がクリアされた場合

$$V = S_m \cdot D_m \cdot \overline{R_m} \quad (6-4)$$

(2) 正の整数と正の整数を加算した結果、符号ビットに1が設定された場合

$$V = \overline{S_m} \cdot \overline{D_m} \cdot R_m \quad (6-5)$$

上記(1)、(2)より加算命令におけるオーバーフローは式(6-6)で表すことができる。

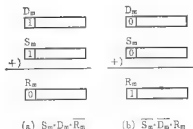


図 6.2 加算命令においてオーバーフローが生じる場合

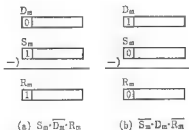


図 6.3 減算、比較命令においてオーバーフローが生じる場合

$$V = S_m \cdot D_m \cdot \overline{R_m} + \overline{S_m} \cdot \overline{D_m} \cdot R_m \quad (6-6)$$

減算命令 (SUB, SUBI, SUBQ, SUBX 命令) および比較命令 (CMP, CMPA, CMPI, CMPM 命令) におけるオーバーフローも、図 6.3 に示すように、加算命令と同様に考えることができる。

減算、比較命令ではデスティネーションオペランドからソースオペランドが減算されることに注意を要する。図 6.3 から減算命令および比較命令を実行したとき V フラグは、

(1) 正の整数 - 負の整数 → 符号ビット = 1 の場合、(2) 負の整数 - 正の整数 → 符号ビット = 0 の場合にセットされる。すなわち次式が成り立つことになる。

$$V = S_m \cdot \overline{D_m} \cdot R_m + \overline{S_m} \cdot D_m \cdot \overline{R_m} \quad (6-7)$$

NEG, NEGX 命令は 0 からデスティネーションオペランドを減算する命令であり、減数がデスティネーションオペランドである点が減算、比較命令と異なる点である。この点に注意して V フラグの設定条件を求めると次式のとおりとなる (詳細省略)。

$$V = D_m \cdot R_m \quad (6-8)$$

DIVS, DIVU 命令では商がオーバーフローしたとき V フラグが設定される。

桁移動命令では ASL ( $r \neq 0$ ) 以外の命令は V フラグを常に 0 とする。ASL 命令 ( $r \neq 0$ ) では符号ビットに変化があった場合に V フラグに 1 が設定される。これには次の 2 とおりの場合がある。ここで  $D_m$  は桁移動前の最上位ビット、 $r$  は桁移動の数である。

(1)  $D_m = 1$  かつ  $D_{m-1} \sim D_{m-r}$  のいずれかが 0 の場合

$$V = D_m \cdot (\overline{D_{m-1}} + \cdots + \overline{D_{m-r}}) \quad (6-8)$$

(2)  $D_m = 0$  かつ  $D_{m-1} \sim D_{m-r}$  のいずれかが 1 の場合

$$V = \overline{D_m} \cdot (D_{m-1} + \cdots + D_{m-r}) \quad (6-9)$$

上記 (1), (2) より、ASL 命令 ( $r \neq 0$ ) では次式に従って V フラグが設定される。

$$V = D_m \cdot (\overline{D_{m-1}} + \cdots + \overline{D_{m-r}}) + \overline{D_m} \cdot (D_{m-1} + \cdots + D_{m-r}) \quad (6-10)$$

この条件は桁移動によってデータの符号が変化した場合をチェックするためのものである。

このほか、命令によって0が設定されたり、未定義であったり、以前の値を保持していたりするが、これらについては表6.2を参照されたい。

### 6.2.4 C フラグ

C (Carry) フラグは命令の実行においてキャリイ/ボローが生じたかどうかを示すフラグであり、キャリイ/ボローが生じた場合1が設定され、生じなかった場合0が設定される。

加算命令においてキャリイが生じるのは図6.4に示す3とおりの場合である。したがって、次式で示される条件によってCフラグが設定される。

$$C = S_m \cdot D_m + D_m \cdot \overline{R_m} + S_m \cdot \overline{R_m} \quad (6-11)$$

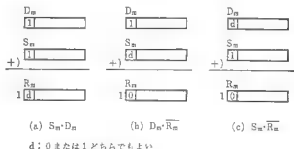


図 6.4 加算命令においてキャリイが生じる場合

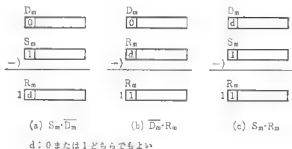


図 6.5 減算、比較命令においてボローが生じる場合

減算、比較命令においてボローが生じるのは図6.5に示す3とりの場合である。したがって、次式で示される条件によって C フラグが設定される。

$$C = S_m \cdot \overline{D_m} + \overline{D_m} \cdot R_m + S_m \cdot R_m \quad (6-12)$$

NEG, NEGX 命令は 0 からデスティネーションオペランドを減算する命令であり、減算がデスティネーションオペランドである点を考慮し、上述した減算、比較命令と同様に考えると次式を得る。

$$C = D_m + R_m \quad (6-13)$$

2 進・10 進数演算命令では 10 進数のキャリイ/ボローが C フラグに反映される。

シフト回数が 0 の桁移動命令において、ROXL, ROXR 命令では C フラグに X フラグの内容がセットされその他のシフト/ローテート命令では C=0 となる。1 回以上シフトまたはローテートする桁移動命令では最後にシフトアウトされたビットが C フラグに反映される。すなわち、右シフトを r 回行った後には

$$C = D_{r-1}$$

となり、左シフトを r 回行った後には

$$C = D_{n-r+1}$$

が設定される。

このほか、C フラグが 0 に設定されたり、未定義であったり、以前の値を保持したりする命令については表 6.2 を参照されたい。

### 6.2.5 X フラグ

X フラグは拡張演算および拡張形桁移動を行うときに使用する。X フラグが変化するときには常に X=C となる。しかし C フラグが変化したとき常に X フラグが影響を受けるわけではない。

X フラグの設定条件は表 6.2 を参照されたい。

# 7 命令実行時間

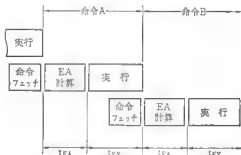
この章では68000の命令実行時間について解説する。

## 7.1 命令実行時間の内訳

一般に命令実行時間は図7.1(a)に示すごとく(1)命令フェッチ時間:  $t_F$ , (2)実効アドレス計算時間:  $t_{EA}$ , (3)命令処理時間:  $t_{EX}$ の総和で表される。ところが68000では、2章にも述べたように、命令プリフェッチ機能を有するために、基本的には命令実行手順



(a) プリフェッチ機能のない場合



(b) プリフェッチ機能のある場合

図 7.1 命令実行時間の内訳

は図 7.1 (b) に示すようになる。すなわち命令フェッチ (時間  $t_F$ ) が、オペランドの実効アドレス計算 (時間  $t_{EA}$ ) または命令処理 (時間  $t_{EX}$ ) と平行して行われるため実効的な命令実行時間は  $t_{EA} + t_{EX}$  で表される。

68000 には、4 章で述べたように、多様なアドレス形式があり、各々のアドレス形式で実効アドレス計算時間  $t_{EA}$  が異なる。また、各命令はその処理時間  $t_{EX}$  が異なるため、次節以降で述べる具体的な命令実行時間では  $t_{EA}$  と  $t_{EX}$  を別々に解説する。また、メモリアクセスは十分高速であるものとし、ウェート状態 (Sw) はないものと仮定した。

## 7.2 実効アドレス計算時間

ここでは時間を ns の代りにクロックサイクル数で示す。クロックサイクル数で表現した理由は、68000 は動作クロック周波数が 4MHz 版から 6MHz~12MHz の品種が用意されているために ns で表現するより一般性を有しているためである。ここでサイクル数は動作クロックの周期を 1 単位として表している。したがって、たとえば 4 クロックサイクルは 8MHz 版 68000 では  $4 \times 125 \text{ ns} = 500 \text{ ns}$  を意味する。

68000 は 4 章で述べたように 12 種の実効アドレス形式をもっている。それら各実効アドレス計算に要するクロックサイクル数は表 7.1 に示される。なお、このサイクル数で処理される内容には次の 2 つが含まれる。

### (1) 実効アドレス計算

表 7.1 実効アドレス計算処理時間 (サイクル数)

ア ド レ ス 形 式	オペランドの長さ	
	バイト/ワード	ロングワード
Dn	データレジスタ直接形式	0 (0/0)
An	アドレスレジスタ直接形式	0 (0/0)
An@	アドレスレジスタ間接形式	4 (1/0)
An@+	ポストインクリメントアドレスレジスタ間接形式	4 (1/0)
An@-	プリデクリメントアドレスレジスタ間接形式	6 (1/0)
An@d	ディスプレースメント付アドレスレジスタ間接形式	8 (2/0)
An@(d, ix)	インデックス付アドレスレジスタ間接形式	10 (2/0)
xxx.W	短絶対アドレス形式	8 (2/0)
xxx.L	長絶対アドレス形式	12 (3/0)
PC@d	ディスプレースメント付プログラムカウンタ相対形式	8 (2/0)
PC@(d, ix)	インデックス付プログラムカウンタ相対形式	10 (2/0)
##xxx	イミディエイト形式	4 (1/0)

注: (a/b), a: リードサイクル数, b: ライトサイクル数

## (2) メモリからのオペランドフェッチ

同表ではオペランドがバイト(8ビット)またはワード(16ビット)の場合とロングワード(32ビット)の場合とに分けて示されている。両者とも実効アドレス計算処理に要するサイクル数は同じであるが、表中で差があるのはオペランドフェッチ処理の差に起因する。すなわち、1ワードのフェッチ(メモリリード)には4サイクルを要しロングワードのフェッチには8サイクル要するためである。また、同表において( ) 中には(リードサイクル数/ライトサイクル数)が示される。これらはいずれも全サイクル数の中に含まれるサイクル数である。実効アドレス計算処理においてライトサイクル数がいずれの場合も0であるのはその処理の性質から当然のことである。

## 7.3 オペレーション実行処理時間

以下に68000の各命令の実行処理時間  $t_{EX}$  をサイクル数で示す。なお、データ MOVE 命令に関してだけは、68000においては種々の実効アドレス形式を有する2つのオペランドを指定し得ることから単純に表7.1に示した  $t_{EA}$  と  $t_{EX}$  を加えた値では表現しにくい。このため MOVE 命令についてはソースオペランドとデスティネーションオペランドの実効アドレス形式の組合せによって処理時間がわかるように  $t_{EA} + t_{EX}$  の全時間で表現した。

## 7.3.1 データ転送命令時間(実効アドレス計算処理時間を含む)

表7.2はオペランドサイズがバイト長またはワード長の場合のデータ転送命令の実行時間を示している。また、表7.3はオペランドサイズがロングワードの場合のデータ転送命令の実行時間を示している。これらの表をよくみると表の対角線を軸としてほぼ対称形の表になっている。しかし  $An@-$  (プリデクリメントアドレスレジスタ間接) モードが指定された場合に限ってこの対称形が保たれない。たとえばソースアドレス =  $An@-$ 、デスティネーションオペランド =  $Dn$  の場合は命令実行時間が10サイクルであるのに対して、逆に、ソースオペランド =  $Dn$ 、デスティネーションアドレス =  $An@-$  の場合は8サイクルですむ。この理由はソースアドレスに  $An@-$  モードが指定された場合は命令実行の最初にアドレスレジスタをデクリメントする2サイクルの操作が入ることによる。デスティネーションオペランドに  $An@-$  モードが指定された場合のアドレスレジスタのデクリメント操作は命令フェッチ中に実行し得るためデクリメント操作のため



表 7.2 オペレーティングシステムがバイト、ワードのときの MOVE 命令実行時間 (サイクル数)

S7アドレス	Dn	An	An@	An@+	An@-	An@(d)	An@(d, ix)	×××, W	×××, L
Dn	4 (1/0)	4 (1/0)	8 (1/1)	8 (1/1)	8 (1/1)	12 (2/1)	14 (2/1)	12 (2/1)	16 (3/1)
An	4 (1/0)	4 (1/0)	8 (1/1)	8 (1/1)	8 (1/1)	12 (2/1)	14 (2/1)	12 (2/1)	16 (3/1)
An@	8 (2/0)	8 (2/0)	12 (2/1)	12 (2/1)	12 (2/1)	16 (3/1)	18 (3/1)	15 (3/1)	20 (4/1)
An@+	8 (2/0)	8 (2/0)	12 (2/1)	12 (2/1)	12 (2/1)	16 (3/1)	18 (3/1)	15 (3/1)	20 (4/1)
An@-	10 (2/0)	10 (2/0)	14 (2/1)	14 (2/1)	14 (2/1)	18 (3/1)	20 (3/1)	18 (3/1)	22 (4/1)
An@(d)	12 (3/0)	12 (3/0)	16 (3/1)	16 (3/1)	16 (3/1)	20 (4/1)	22 (4/1)	20 (4/1)	24 (5/1)
An@(d, ix)	14 (3/0)	14 (3/0)	18 (3/1)	18 (3/1)	18 (3/1)	22 (4/1)	24 (4/1)	22 (4/1)	26 (5/1)
×××, W	12 (3/0)	12 (3/0)	16 (3/1)	16 (3/1)	16 (3/1)	20 (4/1)	22 (4/1)	20 (4/1)	24 (5/1)
×××, L	16 (4/0)	16 (4/0)	20 (4/1)	20 (4/1)	20 (4/1)	24 (5/1)	26 (5/1)	24 (5/1)	28 (6/1)
PC@(d)	12 (3/0)	12 (3/0)	16 (3/1)	16 (3/1)	16 (3/1)	20 (4/1)	22 (4/1)	20 (4/1)	24 (5/1)
PC@(d, ix)	14 (3/0)	14 (3/0)	18 (3/1)	18 (3/1)	18 (3/1)	22 (4/1)	24 (4/1)	22 (4/1)	26 (5/1)
×××	8 (2/0)	8 (2/0)	12 (2/1)	12 (2/1)	12 (2/1)	16 (3/1)	18 (3/1)	16 (3/1)	20 (4/1)

表 7.3 オペレーティングシステムがロングワードの時の MOVE 命令実行時間 (サイクル数)

S7アドレス	Dn	An	An@	An@+	An@-	An@(d)	An@(d, ix)	×××, W	×××, L
Dn	4 (1/0)	4 (1/0)	12 (1/2)	12 (1/2)	12 (1/2)	16 (2/2)	18 (2/2)	16 (2/2)	20 (3/2)
An	4 (1/0)	4 (1/0)	12 (1/2)	12 (1/2)	12 (1/2)	16 (2/2)	18 (2/2)	16 (2/2)	20 (3/2)
An@	12 (3/0)	12 (3/0)	20 (3/2)	20 (3/2)	20 (3/2)	24 (4/2)	26 (4/2)	24 (4/2)	28 (5/2)
An@+	12 (3/0)	12 (3/0)	20 (3/2)	20 (3/2)	20 (3/2)	24 (4/2)	26 (4/2)	24 (4/2)	28 (5/2)
An@-	14 (3/0)	14 (3/0)	22 (3/2)	22 (3/2)	22 (3/2)	26 (4/2)	28 (4/2)	26 (4/2)	30 (5/2)
An@(d)	16 (4/0)	16 (4/0)	24 (4/2)	24 (4/2)	24 (4/2)	28 (5/2)	30 (5/2)	28 (5/2)	32 (6/2)
An@(d, ix)	18 (4/0)	18 (4/0)	26 (4/2)	26 (4/2)	26 (4/2)	30 (5/2)	32 (5/2)	30 (5/2)	34 (6/2)
×××, W	16 (4/0)	16 (4/0)	24 (4/2)	24 (4/2)	24 (4/2)	28 (5/2)	30 (5/2)	28 (5/2)	32 (6/2)
×××, L	20 (5/0)	20 (5/0)	28 (5/2)	28 (5/2)	28 (5/2)	32 (6/2)	34 (6/2)	32 (6/2)	36 (7/2)
PC@(d)	16 (4/0)	16 (4/0)	24 (4/2)	24 (4/2)	24 (4/2)	28 (5/2)	30 (5/2)	28 (5/2)	32 (6/2)
PC@(d, ix)	18 (4/0)	18 (4/0)	26 (4/2)	26 (4/2)	26 (4/2)	30 (5/2)	32 (5/2)	30 (5/2)	34 (6/2)
×××	12 (3/0)	12 (3/0)	20 (3/2)	20 (3/2)	20 (3/2)	24 (4/2)	26 (4/2)	24 (4/2)	28 (5/2)

表 7.4 標準命令実行処理時間 (サイクル数)

命 令	オペランド アドレス オペレー ション サイズ	S アドレス=(EA) D アドレス=A <sub>n</sub>	S アドレス=(EA) D アドレス=D <sub>n</sub>	S アドレス=D <sub>n</sub> D アドレス=(EA)
ADD	バイト, ワード	8 (1/0)	4 (1/0)	8 (1/1)
	ロングワード	11 (1/0)	6 (1/0)	12 (1/2)
AND	バイト, ワード	—	4 (1/0)	8 (1/1)
	ロングワード	—	6 (1/0)	12 (1/2)
CMP	バイト, ワード	6 (1/0)	4 (1/0)	—
	ロングワード	6 (1/0)	6 (1/0)	—
DIVS	—	—	158 (1/0)MAX	—
DIVU	—	—	140 (1/0)MAX	—
EOR	バイト, ワード	—	4 (1/0)	8 (1/1)
	ロングワード	—	8 (1/0)	12 (1/2)
MJLS	—	—	70 (1/0)MAX	—
MJLU	—	—	70 (1/0)MAX	—
OR	バイト, ワード	—	4 (1/0)	8 (1/1)
	ロングワード	—	6 (1/0)	12 (1/2)
SUB	バイト, ワード	8 (1/0)	4 (1/0)	8 (1/1)
	ロングワード	6 (1/0)	6 (1/0)	12 (1/2)

注) 命令実行時間は上表の値に実施アドレス生成時間を加える必要あり  
D=デスティネーション, S=ソース

の2サイクルは実効的な時間としては現れない。

### 7.3.2 標準命令実行時間

表 7.4 に ADD, AND 命令等の標準命令の命令処理時間を示す。これらの命令が使用するアドレス形式は各種あるが、共通していることは (レジスタ, (EA)) 間の処理であるということである。すなわち、データ転送命令のようにソース オペランド、デスティネーション オペランドが共にメモリ上にあることはない、必ず一方はレジスタが指定される。

なお、表 7.4 に示したサイクル数は命令の処理サイクル、結果の格納サイクル、および次の命令のフェッチサイクルを含んでいる。したがって命令の全実行時間を求めるためには (EA) を計算するためのアドレス生成サイクル数  $t_{EA}$  を表 7.1 より求めて表 7.4 の値に加えなければならない。

たとえば次の命令の実行時間を求めてみる。

## ADD .L D1, (A1)

デスティネーションオペランドのアドレス形式はアドレスレジスタ間接形式であり、オペランドサイズがロングワードである。したがって、実効アドレス計算時間  $t_{EA}$  は表 7.1 より 8 クロックサイクルとなる。また、ADD 命令処理時間  $t_{EX}$  は、ソースオペランドが D1、デスティネーションオペランドが (EA)、オペランドサイズがロングワードであるから、表 7.4 より  $t_{EX}=12$  クロックサイクルとなる。以上より上記 ADD 命令の実行時間は  $8+12=20$  クロックサイクルとなる。

## 7.3.3 イミディエイト命令実行時間

表 7.5 にイミディエイト命令の実行処理時間を示す。この表に示される処理時間に含まれる内容はイミディエイトオペランドのフェッチサイクル、命令の実行処理サイクル、結果の格納サイクルおよび次の命令のプリフェッチサイクルである。したがってイミディ

表 7.5 イミディエイト命令実行処理時間 (サイクル数)

命 令	オペランド オペランド サイズ	S アドレス=イミ ディエイト D アドレス=Dn			S アドレス=イミ ディエイト D アドレス=An			S アドレス=イミ ディエイト D アドレス=(EA)		
ADDI	バイト、ワード	8 (2/0)			—			12 (2/1)		
	ロングワード	16 (3/0)			—			20 (3/2)		
ADDQ	バイト、ワード	4 (1/0)			8 (1/0)			8 (1/1)		
	ロングワード	8 (1/0)			8 (1/0)			12 (1/2)		
ANDI	バイト、ワード	■ (2/0)			—			12 (2/1)		
	ロングワード	16 (3/0)			—			20 (3/1)		
CMPI	バイト、ワード	8 (2/0)			8 (2/0)			8 (2/0)		
	ロングワード	14 (3/0)			14 (3/0)			12 (3/0)		
EORI	バイト、ワード	8 (2/0)			—			12 (2/1)		
	ロングワード	16 (3/0)			—			20 (3/2)		
MOVEQ	ロングワード	4 (1/0)			—			—		
ORI	バイト、ワード	■ (2/0)			—			12 (2/1)		
	ロングワード	16 (3/0)			—			20 (3/2)		
SUBI	バイト、ワード	8 (2/0)			—			12 (2/1)		
	ロングワード	16 (3/0)			—			20 (3/2)		
SUBQ	バイト、ワード	4 (1/0)			■ (1/0)			8 (1/1)		
	ロングワード	8 (1/0)			8 (1/0)			12 (1/2)		

注) 命令実行時間は上表の値に実効アドレス生成時間を加える必要あり  
S=ソース、D=デスティネーション

エイト命令の全実行時間を求めるためには (EA) を計算するための実効アドレス計算時間を表 7.1 より求め、表 7.5 の値に加えなければならない。

デスティネーションオペランドが  $D_n$  または  $A_n$  である場合は、デスティネーションオペランドの実効アドレス計算に要する時間は表 7.1 から明らかなように 0 である。したがって表 7.5 に示した値がすなわち命令実行の全時間に等しい。デスティネーションオペランド = (EA) である場合は、種々の実効アドレスモードをとり得るので表 7.1 より実効アドレス生成時間を表 7.5 の値に加えなければならない。

さて、一例として ADDI 命令について述べてみよう。ADDI 命令はイミディエイト値とデスティネーションオペランドの内容を加えて、その結果をデスティネーションアドレスに格納する命令である。オペランドがバイト長またはワード長でデスティネーションオペランドが  $D_n$  の場合を考える。実行処理時間を表 7.5 から求めると 8 (2/0) となっている。すなわち命令処理に 8 サイクルを要する。デスティネーションオペランドは  $D_n$  であるため、そのアドレス生成時間は表 7.1 より 0 サイクルである。すなわち、この命令の全実行時間は  $8+0=8$  サイクルである。

### 7.3.4 シングル オペランド命令実行時間

表 7.6 にシングルオペランド命令の実行処理時間を示す。この表に示される処理時間に含まれる処理内容はオペランドのフェッチ、命令の処理、および次の命令プリフェッチである。したがって命令実行の全時間を求めるためには表 7.1 に示されたオペランドアドレスの計算時間を表 7.6 に示された値に加えなければならない。シングルオペランド命令は CLR, NBCD, NEG 命令など 8 種の命令がある。その特徴はすべて 2 バイト (1 ワード) 命令であること、オペランドには他の命令と異なりデスティネーションオペランドしかないことである。オペランドがレジスタ ( $D_n$  または  $A_n$ ) の場合には、表 7.6 の値に加えるべきアドレス計算時間は表 7.1 より 0 サイクルであるため、表 7.6 の値がそのまま命令の全実行時間となる。一方、オペランドがメモリの場合には、命令の全実行時間は表 7.6 の値に表 7.1 に示したアドレス計算時間を各々実効アドレス形式に従って求めて加えなければならない。

### 7.3.5 シフト/ローテート命令実行時間

表 7.7 にシフト/ローテート命令の実行処理時間を示す。この表に示す処理時間に含まれる内容はオペランドフェッチ、シフト/ローテートの処理、結果の格納および、次の命

表 7.6 シングルオペランド命令実行処理時間 (サイクル数)

命 令	オペランド		オペランド=レジスタ (Dn または An)	オペランド=メモリ (EA)
	オペランド サイズ			
CLR	バイト, ワード		4 (1/0)	■ (1/1)
	ロングワード		6 (1/0)	12 (1/2)
NBCD	バイト		6 (1/0)	8 (1/1)
NEG	バイト, ワード		4 (1/0)	8 (1/1)
	ロングワード		6 (1/0)	12 (1/2)
NEGX	バイト, ワード		4 (1/0)	■ (1/1)
	ロングワード		6 (1/0)	12 (1/2)
NOT	バイト, ワード		4 (1/0)	8 (1/1)
	ロングワード		6 (1/0)	12 (1/2)
Scc	バイト, ワード		4 (1/0)	■ (1/1)
	ロングワード		■ (1/0)	8 (1/1)
TAS	バイト		4 (1/0)	10 (1/1)
TST	バイト, ワード		4 (1/0)	4 (1/0)
	ロングワード		4 (1/0)	4 (1/0)

注) 命令実行時間は上表の値に実効アドレス生成時間を加える必要あり

表 7.7 シフト/ローテート命令実行処理時間 (サイクル数)

命 令	オペランド		オペランド=レジスタ	オペランド=メモリ (EA)
	オペランド サイズ			
ASR, ASL	バイト, ワード		$6+2n^*$ (1/0)	8 (1/1)
	ロングワード		$8+2n$ (1/0)	—
LSR, LSL	バイト, ワード		$6+2n$ (1/0)	8 (1/1)
	ロングワード		$8+2n$ (1/0)	—
ROR, ROL	バイト, ワード		$6+2n$ (1/0)	8 (1/1)
	ロングワード		$8+2n$ (1/0)	—
ROXR, ROXL	バイト, ワード		$6+2n$ (1/0)	8 (1/1)
	ロングワード		$8+2n$ (1/0)	—

注) 命令実行時間は上表の値に実効アドレス生成時間を加える必要あり

\* n は桁数の値を表す

令のブリフエッチである。したがってオペランドアドレスの計算時間を表7.1により求め、表7.7に示した値に加えて命令の全実行時間を求めなければならない。シフト/ローテート命令の機能はオペランドがレジスタの場合には1命令でnビットのシフトまたはローテートが可能であるが、一方オペランドがメモリの場合には1命令で実行し得るシフ

ト/ローテートビット数は1ビットだけである。1ビットのシフトには2クロックサイクルを必要とすることから、表7.7においてオペランドがレジスタの場合には+2nサイクルが加えられている。

たとえば次の命令の実行時間は、 $t_{RA}=0$ 、 $t_{RX}=6+2\times 8=22$ であるから22クロックサイクルとなる。

ASL .W #8, D1

### 7.3.6 ビット操作命令実行時間

表7.8にビット操作命令の実行処理時間を示す。この表に示す値はオペランドフェッチ、命令の処理、結果の格納。および次の命令のプリフェッチを含んでいる。したがって命令の全実行時間を求めるためにはオペランドアドレスの計算時間を表7.1から求めて表7.8の値に加えなければならない。表7.8中のダイナミック形とは操作すべきオペランドのビット位置が命令で指定されるレジスタに貯えられている場合を示し、一方スタティック形とは操作すべきオペランドのビット位置が直接命令で指定される場合を示している。

### 7.3.7 ブランチ関係命令実行時間

表7.9にブランチ関係の命令実行処理時間を示す。この表には、ブランチ先のアドレス(ディスプレースメント付プログラムカウンタ相対形式)の計算に要する時間が含まれている。したがってBcc, BRA, BSR, DBcc 命令に関しては本表に示された値が命令の全実行時間を示している。また TRAP, TRAPV 命令についてもベクタ生成時間は本表の値に含まれているので命令の全実行時間を示している。しかしCHK 命令ではブランチ先のアドレスとして各種の実効アドレス形式を有しているので命令の全実行時間を求めるためには表7.9に示される値に、表7.1より求めた実効アドレス計算時間を加えなければならない。さて、表7.9はブランチが成立した場合と不成立の場合について分けて示している。BRA, BSR, TRAP 命令は無条件ブランチであるので不成立の欄は無意味である。68000では命令のプリフェッチ機能を有しているため、ブランチが成立したときには先行してプリフェッチしていた命令が無駄になる。したがってブランチ条件が成立した場合には命令処理時間がブランチ条件不成立の場合に比べて長くなると考えられる。この状況が表7.9のBcc 命令、ディスプレースメント=バイトの場合の実行時間に現れている。しかし、この弊害を最小化するために68000ではブランチアドレスをマイクロプロ

表 7.8 ビット操作命令実行処理時間 (サイクル数)

命 令	ビット位置		ダイナミック		スタティック	
	オペランドサイズ	オペランド	レジスタ	メモリ(EA)	レジスタ	メモリ(EA)
BCHG	バイト	—	—	8 (1/1)	—	12 (2/1)
	ロングワード	8(1/0)MAX	—	—	12(2/0)MAX	—
BCLR	バイト	—	—	8 (1/1)	—	12 (2/1)
	ロングワード	10(1/0)MAX	—	—	14(2/0)MAX	—
BSET	バイト	—	—	8 (1/1)	—	12 (2/1)
	ロングワード	8(1/0)MAX	—	—	12(2/0)MAX	—
BTST	バイト	—	—	4 (1/0)	—	8 (2/0)
	ロングワード	6(1/0)MAX	—	—	10(2/0)MAX	—

注) 命令実行時間は上表の値に実効アドレス生成時間を加える必要あり

表 7.9 ブランチ関係命令実行処理時間 (サイクル数)

命 令	条件成立の有無		トラップまたはブランチ条件成立	トラップまたはブランチ条件不成立
	ディスプレイメント			
Bcc	バイト		10 (2/0)	8 (1/0)
	ワード		10 (2/0)	12 (2/0)
BRA	バイト		10 (2/0)	—
	ワード		10 (2/0)	—
BSR	バイト		18 (2/2)	—
	ワード		18 (2/2)	—
DBcc	CC true		—	12 (2/0)
	CC false		10 (2/0)	14 (3/0)
CHK	—		40 (5/3)+tRA MAX	10 (1/0)+tRA
TRAP	—		34 (4/3)	—
TRAPV	—		34 (5/3)	4 (1/0)

注) 命令実行時間は上表の値に実効アドレス生成時間を加える必要あり

ムで先行して計算するように配慮されているので、場合によってはブランチ条件が成立した場合の方が不成立の場合より高速に処理できるのである。その例が Bcc 命令においてディスプレースメント=ワードの場合である。Bcc 命令でディスプレースメントがワード長の場合にはロングワード命令(4バイト命令)となるためブリフエッチの効果が生きないことがその理由である。DBcc 命令の実行時間がブランチ条件成立の場合が不成立の場合よりも高速になっているのも同じ理由による。

表 7.10 ジャンプ関係命令実行時間 (サイクル数)

E. A. 命令	An@	An@+	An@-	An@ (d)	An@ (d, ix)	×××.W	×××.L	PC@ (d)	PC@ (d, ix)
JMP	8(2/0)	—	—	10(2/0)	14(3/0)	10(2/0)	12(3/0)	10(2/0)	14(3/0)
JSR	16(2/2)	—	—	18(2/2)	22(2/2)	18(2/2)	20(3/2)	18(2/2)	22(2/2)

E. A.=実効アドレス形式

表 7.11 実効アドレス生成関係命令実行時間 (サイクル数)

E. A. 命令	An@	An@+	An@-	An@ (d)	An@ (d, ix)	×××.W	×××.L	PC@ (d)	PC@ (d, ix)
LEA	4(1/0)	—	—	8(2/0)	12(2/0)	8(2/0)	12(3/0)	8(2/0)	12(2/0)
PEA	12(1/2)	—	—	16(2/2)	20(2/2)	16(2/2)	20(3/2)	16(2/2)	20(2/2)

E. A.=実効アドレス形式

## 7.3.8 ジャンプ関係命令実行時間

表 7.10 にジャンプ関係命令 (JMP, JSR 命令) の実行時間を示す。ここに示された値はジャンプ先アドレスの計算、ジャンプの実行およびジャンプ先番地に格納されている次の命令プリフェッチの時間を含んでいる。すなわち、実効アドレス計算時間が表 7.10 に含まれているため表 7.1 に示した実効アドレス計算時間を加えなくても、表 7.10 の値はジャンプ命令の全実行時間を表している。

## 7.3.9 実効アドレス生成関係命令実行時間

表 7.11 に実効アドレス生成関係命令 (LEA, PEA 命令) の実行時間を示す。ここに示された値には実効アドレス計算時間が含まれるので改めて表 7.1 の値を加える必要はない。すなわち、表 7.11 の値は命令の全実行時間を表している。

## 7.3.10 マルチレジスタ関係命令実行時間

表 7.12 にマルチレジスタ関係命令の実行時間を示す。マルチレジスタ関係の命令は MOVEM 命令のみである。同表において n は移動すべきオペランドの数を示している。ワード長オペランドのときにはメモリーレジスタ間移動に要するサイクル数は 1 オペランドにつき 4 サイクル、一方ロングワードオペランドのときには 1 オペランド当り 8 サイクルを要する。表 7.12 に示される値には実効アドレス生成のための時間が含まれているので、改めて表 7.1 から実効アドレス生成時間を求め表 7.12 の値に加える必要はない。すなわち、表 7.12 の値は MOVEM 命令の全実行時間を表している。M→R の場合と R→M



表 7.12 マルチレジスタ関係命令実行時間 (サイクル数)

命令	E. A. O. S.	An@	An@÷	An@-	An@ (d)	An@ (d, ix)	×××.W	×××.L	PC@ (d)	PC@ (d, ix)
MOVEM (M→R)	ワード	12+4n (3+n/0)	12+4n (3+n/0)	—	16+4n (4+n/0)	18+4n (4+n/0)	16+4n (4+n/0)	20+4n (5+n/0)	16+4n (4+n/0)	18+4n (4+n/0)
	ロング ワード	12+8n (3+2n/0)	12+8n (3+n/0)	—	16+8n (4+2n/0)	18+8n (4+2n/0)	16+8n (4+2n/0)	20+8n (5+2n/0)	16+8n (4+2n/1)	18+8n (4+2n/0)
MOVEM (R→M)	ワード	8+4n (4/n)	—	8+4n (2/n)	12+4n (3/n)	14+4n (3/n)	12+4n (3/n)	16+4n (4/n)	—	—
	ロング ワード	8+8n (2/2n)	—	8+8n (2/2n)	12+8n (3/2n)	14+8n (3/2n)	12+8n (3/2n)	16+8n (4/2n)	—	—

O. S.=オペランドサイズ, E. A.=実効アドレス形式

表 7.13 倍精度演算命令実行時間 (サイクル数)

命令	O. O. O. S.	S オペランド=Dn D オペランド=Dn	S オペランド=メモリ D オペランド=メモリ
ADDX	バイト, ワード ロングワード	4 (1/0) 8 (1/0)	18 (3/1) 30 (5/2)
SUBX	バイト, ワード ロングワード	4 (1/0) 8 (1/0)	18 (3/1) 30 (5/2)
ABCD	バイト	6 (1/0)	18 (3/1)
SBCD	バイト	6 (1/0)	18 (3/1)
CMPM	バイト, ワード ロングワード	— —	12 (3/0) 20 (5/0)

O. S.=オペランドサイズ, O. O.=演算オペランド形式, S=ソース, D=デスティネーション

の場合の実行時間に4サイクルの差がある。たとえばワードオペランド、実効アドレス=An@の場合を考えるとM→Rのときは(12+4n)サイクルであるが、R→Mのときは(8+4n)サイクルである。4nサイクルはnワード分のメモリリードまたはライトに要するサイクル数である。差の4サイクル分(12-8=4)は68000のアーキテクチャにかかわるもので、高速動作実現のために68000はメモリデータの先読みを実行しており、nワード移動に対して実際はn+1ワード分メモリからMPU内にリードしていることによるものである。

## 7.3.11 倍精度演算命令実行時間

表 7.13 に倍精度演算命令の実行時間を示す。表 7.13 の示す値は 2つのオペランド フェッチ、演算、結果の格納、および次の命令のとり込みの各時間のほかに 2つのオペランドの実効アドレス計算時間を含んでいる。オペランドの実効アドレスの生成時間を含めて表現した理由はこれらの命令では 2つのメモリ オペランドが扱われるからである。表 7.1 に示したオペランドの実効アドレス生成時間は 1つのオペランドアドレスを計算する時間を表している。2つのメモリ オペランドを有する命令の実行時間は表 7.1 に示されたメモリ オペランドが 2つあるから単純に 2倍して実行処理時間に加えるという計算で簡単に求めることはできない。

## 7.3.12 その他の命令実行時間

表 7.14 に前項までに述べた各種命令以外の残りの命令の実行時間をまとめて示す。表

表 7.14 その他の命令実行処理時間 (サイクル数)

命 令	O. O.S.	O.			
		レジスタ	メモリ	レジスタ→メモリ	メモリ→レジスタ
MOVE from SR	—	6 (1/0)	11 (1/1)	—	—
MOVE to CCR	—	12 (2/0)	12 (2/0)	—	—
MOVE to SR	—	12 (2/0)	12 (2/0)	—	—
MOVEP	ワード	—	—	16 (2/2)	16 (4/0)
	ロングワード	—	—	24 (2/4)	24 (6/0)
EXG	—	6 (1/0)	—	—	—
EXT	ワード	4 (1/0)	—	—	—
	ロングワード	4 (1/0)	—	—	—
LINK	—	16 (2/2)	—	—	—
MOVE from USP	—	4 (1/0)	—	—	—
MOVE to USP	—	4 (1/0)	—	—	—
NOP	—	4 (1/0)	—	—	—
RESET	—	132 (1/0)	—	—	—
RTE	—	20 (5/0)	—	—	—
RTR	—	20 (5/0)	—	—	—
RTS	—	16 (4/0)	—	—	—
STOP	—	4 (0/0)	—	—	—
SWAP	—	4 (1/0)	—	—	—
UNLK	—	12 (3/0)	—	—	—

注) 命令実行時間は上表の値に実効アドレス生成時間を加える必要あり  
O.S.=オペランドサイズ, O.=オペランド形式

7.14に示した値にはオペランドの実効アドレス生成時間は含まれていないので、命令実行時間を求めるためには表7.1に示した実効アドレス生成時間を加えなければならない。オペランドがレジスタのみの場合の実効アドレス生成時間は0であるから実質的には表7.1に示した値が実行時間になる。しかし、オペランドがメモリの場合は表7.1から実効アドレス生成時間を求めなければならない。MOVE from SR, MOVE to SR, MOVE to CCRの各命令はメモリオペランドとして各種実効アドレスモードを指定し得る。MOVEP命令については指定されるメモリオペランド実効アドレス形式はAn@(d)形式のみである。

### 7.3.13 例外処理時間

表7.15に例外処理に必要なクロック数を示す。例外処理は命令の実行ではないがその処理時間を示しておく。例外処理の内容については8章で説明する。表7.15で示されるクロックサイクル数はプロセッサが例外処理を開始してから例外処理の新プログラムの跳出しまでの時間である。

表 7.15 例外処理時間 (サイクル数)

例外処理	時 間
リセット	34 (6/0)
アドレス エラー	50 (4/7)
バス エラー	50 (4/7)
割込み	44 (5/3)*
不当命令	34 (4/3)
特権命令	34 (4/3)
トレース	34 (4/3)

\* 割込みアタリレジスタバスサイクルには、4クロックサイクルを要すると仮定している

## 8 例外処理

この章では 68000 のプロセッサ処理状態、プログラム実行状態、例外処理について説明する。これらは他のマイクロコンピュータにはみられない進んだアーキテクチャであり、68000 を理解する上で重要である。

### 8.1 プロセッサ処理状態

68000 が動作をしている場合、その動作は通常状態、例外状態、ホールド状態の 3 状態に分類される。これらの状態を総称してプロセッサ処理状態という。

通常状態とはプログラムを実行している状態である。この状態では、メモリ参照によって命令やオペランド等を読み出し、演算を行った後にその結果を内部レジスタあるいはメモリに格納する。これらの動作は連続的に実行されるが、その例外として STOP 命令 (17 章、特権命令参照) の実行によるストップ状態がある。STOP 命令を実行すると、この命令によってステータスレジスタにセットされた割込みレベルより高いレベルの外部割込みが発生するまで、プロセッサはストップする。このストップ状態は STOP 命令を正常に実行した結果である。そのため、後述するホールド状態のようなシステムの回路要求すなわち、外部からの制御信号の入力による停止とは区別され、通常状態として分類する。

例外状態とは、外部割込み、トラップ命令の実行、バスエラー、アドレスエラー等の例外処理要因によって起動される処理を実行している状態である。例外処理で実行される動作に関しては 8.3 節で詳細に説明する。

ホールド状態とはシステムの回路的要求によってプロセッサが停止する状態である。この回路的要求には、3 章でも説明したようにホールド信号入力ピンのアサートによる場合と 2 重バス障害による場合とがある。前者は、68000 に接続された外部デバイスが何らかの重大な故障を起した場合、ホールド信号を入力してプロセッサを停止させる場合であ

る。後者は、例外処理実行中にメモリ参照ができないためその処理が続行できないという重大な故障が発生した場合、プロセッサを停止させるものである。

プロセッサ処理状態の遷移を図8.1にまとめる。通常状態から例外状態への遷移は、前述のように例外処理を必要とする要因の発生によって行われる。その例外処理が終了すると再び通常状態に遷移する。通常状態、例外

状態においてハールド信号入力ピンがアサートされるとハールド状態に遷移する。ハールド信号入力ピンをネゲートすると、ハールド状態となる前の状態へ遷移する。また、例外処理中にメモリの参照が不可能となり2重バス障害（通常状態では2重バス障害は発生しない）が発生した場合には、例外状態からハールド状態に遷移する。その後、プロセッサを再び起動させるためには、RES ピンをアサートして、例外状態へ遷移させる必要がある。

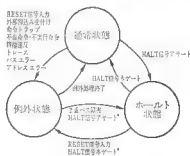


図 8.1 プロセッサの処理状態の遷移

## 8.2 プログラム実行状態

68000 では、前節8.1で説明した通常状態において、実行するプログラムを一般のプログラムとシステムプログラムとに区別できるようにしている。そのためにプログラム実行状態<sup>†</sup>をユーザ状態とスーパーバイザ状態という2種類の状態に区別する。この2種類の状態は、ステータスレジスタ内のSビット（ビット番号13）によって選択される。すなわちSビットが“0”の場合にはユーザ状態であり、“1”の場合にはスーパーバイザ状態である。

システムの信頼性を考えた場合、一般のプログラムの実行と、システムを管理するプログラム（OSと称されるシステムプログラム）の実行とを区別することが重要である。一般のプログラムの実行において、システム全体に影響を与えるような動作が簡単にに行われると、いくつかのプログラムを並列させて実行する場合には大きな障害となりがねない。そこで、一般のプログラムで実行できる機能は制限し、システムを管理する機能等は特殊な状態で実行するシステムプログラムに任せる方式がシステム全体の信頼性を向上させるためには有効である。

† 従来の68000解説書ではプログラム実行状態（Privilege State）を特権状態と訳してきた。しかし、この特権状態という表現はスーパーバイザ状態（この状態においてのみ特権命令を使用できる）と混同される可能性があるため、本書では、プログラム実行状態という表現を用いることにした。

スーパーバイザ状態は、プログラム実行状態の2つの状態のうちの上位の状態に相当する。一般にこの状態でシステムプログラムが実行される。システムを管理する上で有効な命令である特権命令（17章参照）は、スーパーバイザ状態においてのみ使用可能である。

一方、ユーザ状態は、プログラム実行状態における下位の状態に相当する。この状態で前述の特権命令を使用しようとする、その実行の段階で特権違反という例外処理（8.3節参照）が発生する。

プログラム実行状態の遷移を図8.2にまとめる。ユーザ状態からスーパーバイザ状態への遷移は、図8.1に示したプロセッサ処理状態における通常状態から例外状態への遷移が行われるのと同時に行われる。すなわち、例外状態で実行される例外処理はスーパーバイザ状態

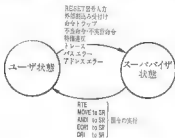


図 8.2 プログラム実行状態の遷移

を用い、ステータスレジスタ内のSビットを“0”に書き換えればよい。

プログラム実行状態による区別は、特権命令の使用を規定するのみでなく、命令オペランド等の情報の参照にも影響を与える。たとえば68000の内部レジスタであるアドレスレジスタA7は、システムスタックポインタとしても用いられる。このA7として指定されるアドレスレジスタ（システムスタックポインタ）は2本存在する。一方はユーザ状態で指定されるユーザシステムスタックポインタ（USP）であり、もう一方がスーパーバイザ状態で指定されるスーパーバイザシステムスタックポインタ（SSP）である。プログラム実行状態によって、どちらのレジスタを使用するのかが自動的に決められる。

メモリ参照の際にも、プログラム実行状態を区別するため、アドレス信号の出力（A1～A23および $\overline{\text{LDS}}$ 、 $\overline{\text{UDS}}$ ）と同期してファンクションコード（FC0～2）が出力される。ファンクションコードによる分類はすでに表3.2に示した。FC2がプログラム実行状態を表し、FC0、FC1でプログラム参照、データ参照の区別を表している。この信号を有効に利用することによって、ユーザ状態でのプログラム実行中に、スーパーバイザ領域のデータの参照を制限することができる。

態で実行される。例外処理終了後、プロセッサ処理状態が通常状態へと遷移してもステータスレジスタ内のSビットは、まだ“1”のままである。そのため、例外処理後のプログラムは、まずスーパーバイザ状態から開始される。プログラム実行状態をスーパーバイザ状態からユーザ状態に遷移させるには、スーパーバイザ状態でのみ使用が許されている特権命令（図8.2に示す）

### 8.3 例外処理

8.1 および 8.2 冊で説明したように、外部割込み、TRAP 命令等により、プロセッサ処理状態は例外状態となり、プログラム実行状態はスーパーバイザ状態へと遷移する。本節では、この例外状態におけるプロセッサの動作（以後、例外処理と表記）について説明する。

まず例外処理の概要を、図 8.3 を用いて説明する。例外処理は 5 種類の動作（分類番号Ⅰ～Ⅴ、各要因別の説明のためにフローチャート内に分類番号を記載した）に大別される。これらの動作は、例外処理発生要因ごとに多少異なるので、概要説明後に要因別の詳細説明を行う。

例外処理を必要とする要因が発生すると、以下に述べる例外処理が開始される。Ⅰでは、まず例外処理を行う前のステータスレジスタの内容を 68000 内システムレジスタ（ユーザが直接アクセスすることはできないレジスタ）に退避する。その後、ステータスレジスタの S ビット（ビット番号 13）を“1”にして、スーパーバイザ状態に移行させる。同時に、例外処理終了後に起動されるプログラムでのトレースを抑制するため、T ビット（ビット番号 15）を“0”とする。

Ⅱでは例外ベクタ（例外処理終了後に起動されるプログラムのアドレス）を獲得するための準備が行われる。すなわち例外処理を発生させた要因に従って 68000 内で自動生成させた例外ベクタ番号（たとえば、バスエラーの場合の例外ベクタ番号は 2）、あるいは外部デバイスが 68000 に知らせた例外ベクタ番号（101 ページ、割込み例外処理参照）を 4 倍して、例外ベクタのアドレスとする。例外処理発生要因と、例外ベクタ番号および例外ベクタアドレスの関係を表 8.1 にまとめる。ベクタ番号 64～255 が、一般の外部デバイスから 68000 に要因を知らせるために用意されている例外ベクタ番号である。

Ⅲでは、例外処理を行う前の 68000 内部状態をメモリに退避する。スーパーバイザシステムスタック（SSP が指定するアドレスのメモリ上）にプログラムカウンタの内容（この値は、ここまでの例外処理では変化しない）および例外処理を行う前のステータスレジスタの内容（Ⅰでシステムレジスタに退避した値）を格納する。これらの格納は、最初スーパーバイザスタックポインタが指定していたアドレスからマイナス 2 バイトした位置を先頭としてプログラムカウンタの下位 2 バイト分を、さらにアドレスが減少する方向に、プログラムカウンタの上位 2 バイト分、システムレジスタに退避されたステータスレジ

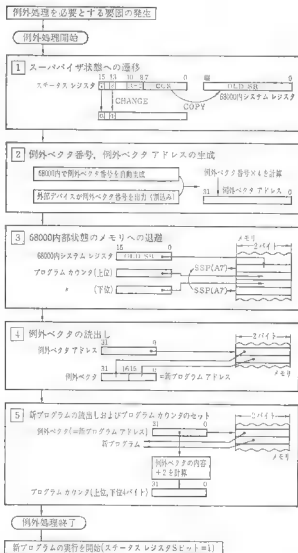


図 8.3 例外処理の概要

タの内容を書き込んでいく。書込みが終了した時点では、スタックポインタの内容は、書込みを行う前の値から 6 マイナスした値 (バスエラー、アドレスエラーでは 14 マイナスした値) となっている。



表 8.1 例外ベクタの割当て

Vector Number(s)	Address			Assignment
	Dec	Hex	Space	
0	0	000	SP	Reset: Initial SSP
1	4	004	SP	Reset: Initial PC
2	8	008	SD	Bus Error
3	12	00C	SD	Address Error
4	16	010	SD	Illegal Instruction
5	20	014	SD	Zero Divide
6	24	018	SD	CHK Instruction
7	28	01C	SD	TRAPV Instruction
8	32	020	SD	Privilege Violation
9	36	024	SD	Trace
10	40	028	SD	Line 1010 Emulator
11	44	02C	SD	Line 1111 Emulator
12*	48	030	SD	(Unassigned, Reserved)
13*	52	034	SD	(Unassigned, Reserved)
14*	56	038	SD	(Unassigned, Reserved)
15	60	03C	SD	Uninitialized Interrupt Vector
16~23*	64	04C	SD	(Unassigned, Reserved)
	95	05F		
24	96	060	SD	Spurious Interrupt
25	100	064	SD	Level 1 Interrupt Auto-vector
26	104	068	SD	Level 2 Interrupt Auto-vector
27	108	06C	SD	Level 3 Interrupt Auto-vector
28	112	070	SD	Level 4 Interrupt Auto-vector
29	116	074	SD	Level 5 Interrupt Auto-vector
30	120	078	SD	Level 6 Interrupt Auto-vector
31	124	07C	SD	Level 7 Interrupt Auto-vector
32~47	128	080	SD	TRAP Instruction Vectors
	191	0BF		
48~63*	192	0C0	SD	(Unassigned, Reserved)
	255	0FF		
64~255	256	100	SD	User Interrupt Vectors
	1023	3FF		

SP: Supervisor Program, SD: Supervisor Data

\* ベクタ番号 12, 13, 14, 16~23, 48~63 は将来使用する可能性がある。ユーザの周辺 LSI にこれらの番号を割り当ててはならない。

④では、②で生成した例外ベクタのアドレスを用いて、例外処理終了後に実行する新プログラムのアドレス（例外ベクタ）を読み出す。

そして⑥で、この例外ベクタとして読み出した新プログラムのアドレスを用いて、新プ

表 8.2 例外処理の分類と優先度

グループ	例外処理の種類	例外処理発生要因の検出	例外処理の実行
0	リセット アドレス エラー バス エラー	クロック サイクル毎に検出	次のマイナサイクル(2クロック サイクル単位)から実行
1	トレース  割込み 不当命令 未実行命令 特権違反	命令サイクルの最後で検出  その直前に実行中の命令サイクル内で検出	次の命令の実行直前に実行  その命令実行の直前に実行
2	命令トラップ (TRAP, TRAPV, CHK, 除数0)	その命令の命令サイクル内で検出	その命令実行の途中から実行

■ グラムを4バイト読み出す。同時に、プログラムカウンタの内容を、例外ベクタで指定されたアドレス値に2を加えた値とする。

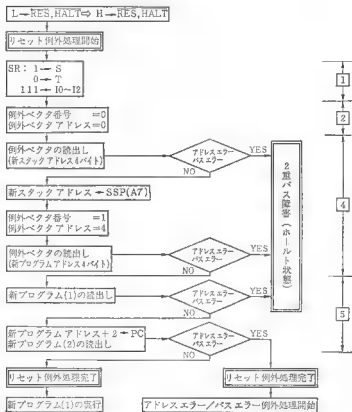
以上で例外処理は完了し、図で読み出した新プログラム（エラー回復処理を行うシステムプログラム）の実行を開始する。なお、ステータスレジスタのSビットは“1”のままであるから、プログラム実行状態はスーパーバイザ状態である。

複数の例外処理発生要因が同時に示された場合には、表8.2に説明する優先度に従って、1つずつ順番に処理される。例外処理発生要因の検出タイミングには3種類あり、1クロック毎に検出するものをグループ0、次の命令を実行する前に検出するものをグループ1、命令実行のシーケンス内で検出するものをグループ2と分類する。表8.2は、この分類に従ってグループ分けした例外処理発生要因である。これらグループ間での例外処理優先度は、グループ0が一番高く、グループ2が一番低い。そのため、たとえばステータスレジスタのTビットが1となっていてトレース例外処理を要求する状態であっても、バスエラーが発生した場合には、まず、グループ0のバスエラー例外処理が実行され、そのエラー処理（例外処理およびそれに続くエラー回復用のシステムプログラムの実行）が終了してから、トレース例外処理を実行することになる（ただし、前述システムプログラムの最後で、バスエラー例外処理を行う前のステータスレジスタの内容を再びプロセッサ内のステータスレジスタに格納しないと、トレース例外処理は発生しない）。

また、グループ内にも優先度がある。グループ0では、リセット例外処理の優先度が最も高く、バスエラー例外処理の優先度が最も低い。そのため、同時にバスエラーとアドレスエラーが発生した場合には、まず、アドレスエラー例外処理が行われる。グループ

1での優先度は、高い方から、トレース■例外処理、誤込み例外処理、不当命令・未実装命令の例外処理、特権違反例外処理の順である。しかし、グループ2内では優先度という考え方はない。これは、グループ2の例外処理の発生はある特定の命令の実行に起因しており、これらの要因が同時に発生することはないからである。

a. リセット例外処理 外部リセットピン入力のアサートによって発生する例外処理をリセット例外処理という。リセット例外処理は、システムの初期状態設定および重大な故障（2重バスエラー等）からの回復を■するためのものであり、最も優先度の高い例外処理である。これによってシステム全体がリセットされる。一方、特権命令として提供されている RESET 命令は、68000に接続された外部デバイスをリセットするためのものである。そのため、この特権命令を実行すると、外部リセットピン出力はアサートされるが、



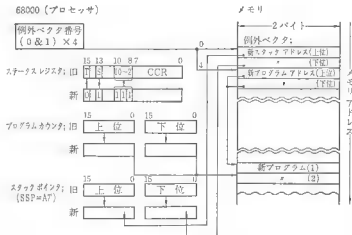


図 8.3 リセット例外処理

例外処理は行われない (17 章、特権命令参照)。

リセット例外処理の詳細を、図 8.4 および図 8.5 を用いて説明する。図 8.4 の右側に示した分類番号Ⅰ～Ⅳは、例外処理の概要を説明した図 8.3 の分類番号Ⅰ～Ⅳに対応している。例外処理が開始されると、まず、スーパーバイザ状態への遷移が行われる。リセット例外処理では、例外処理を行う前の 68000 内部状態を記録しておく必要がないので、ステータスレジスタのシステムレジスタへの一時退避は行われない。

次に、例外ベクタ番号および例外ベクタアドレスの生成が行われる。

最初に生成される例外ベクタ番号 0 (例外ベクタアドレス=0) からの 4 バイトにはスーパーバイザスタックポインタの初期値 (スタックアドレス) が格納されている。その新スタックアドレスは、メモリより読み出され、スーパーバイザスタックポインタ (SSP=A7) にセットされる。その後、例外ベクタ番号 1 (例外ベクタアドレス=4) からの 4 バイトに格納されている新プログラムアドレスがメモリより読み出される。

8 バイトの例外ベクタ 0 番と 1 番の読出しが終了すると、リセット例外処理終了後に実行される新プログラムの読出しが行われる。この新プログラムの先頭アドレスとして、例外ベクタ 1 番から読み出された例外ベクタの内容が用いられる。この先頭アドレス (例外ベクタ) からの 4 バイトが、新プログラムとして読み出される。さらにプログラムカウンタには、例外ベクタから読み出された先頭アドレスに 2 を加えた値がセットされる。

2 重バース障害の発生する可能性があるのは、例外ベクタ読出しおよび新プログラムの最



い、以下、割込み例外処理の詳細について説明する。

割込み要求が受け付けられると、図 8.6、8.7 に示す割込み例外処理が行われる。まず、例外処理の概要で説明したスーパーバイザ状態への遷移①が実行される。このとき、外部割込みレベル入力ピンで指定した割込みレベルを、ステータスレジスタ割込みマスク情報 I0~I2 にセットする。次いで、現状プログラムカウンタの内容から 2 を引いた値の下位

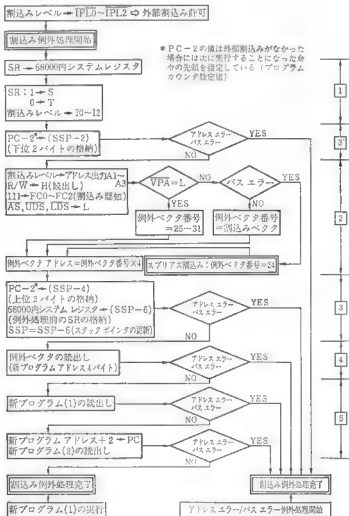


図 8.6 割込み例外処理シーケンス

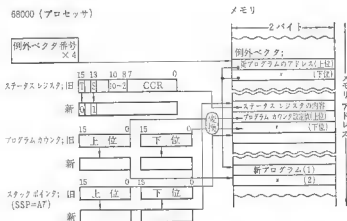


図 8.7 割込みトレース、不当命令、未実装命令、特権違反、命令トラップの例外処理

2 バイト分が、スーパーバイザスタック ポインタ (SSP=A7) によってアドレスを指定されたメモリ上 (以後、“スーパーバイザシステムスタック上”と表記) に退避される。現状プログラムカウンタの内容から 2 を引いた値は、例外処理が発生しなかった場合に次に実行する命令の先頭アドレスを表している。

割込み例外処理の特徴は、例外ベクタ番号の発生法である。割込み例外処理のタイミングチャートを図 8.8 に示すが、例外ベクタ番号の発生は割込みアクノレッジサイクルで行われる。割込みアクノレッジサイクルではまず、アドレス出力ピン A1~3 に割込みレベルをセットし、同時にファンクションコード出力ピン FC0~2 に割込み認知を表す“111”をセットして、あたかもメモリの読出しを行うかのように動作する。この読出し動作に対して、割込みを認められた外部デバイスが応答する。この応答で、 $\overline{\text{VPA}}$  入力ピンを“Low”レベルにセットすると、他の例外処理と同様に、例外ベクタ番号は 68000 内で自動生成される (表 8.1 例外ベクタ番号 25~31)。一方、 $\overline{\text{VPA}}$  入力ピンを“High”レベルにした場合は、外部デバイスがデータバスを通して 68000 に例外ベクタ番号をセットする (68000 はデータバスから読込みを行う)。データバス上に示される割込みベクタの形式を図 8.9 に示す。ここで割込みベクタとして指定する例外ベクタ番号は、表 8.1 で示したように 64~255 のどれか 1 つの値とする (ハード的には 0~255 を指定できるが、すでに例外ベクタ番号として割り当てられているので、割込みベクタ番号として、使用することは好ましくない)。この外部デバイスからの割込みベクタとして指定される例外ベクタ番

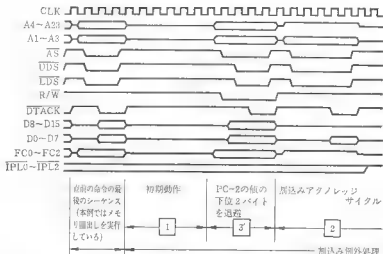


図 8.8 割込み例外処理のシーケンス タイミング

D15	8	7	6	5	4	3	2	1	0
無	視	V7	V6	V5	V4	V3	V2	V1	V0
外部割込み時に設定される値		0	1	0	0	0	0	0	0 (=64)
読みベクタ(例外ベクタ番号)		1	1	1	1	1	1	1	1 (=255)

注) 読みベクタとして0~63の例外ベクタ番号を設定することも可能である。  
しかし、8~63の例外ベクタ番号の使用はある機能を予期して設定されたものであるため、読みベクタ番号として使用することは好ましくない。

図 8.9 割込み例外処理で外部デバイスが出力する割込みベクタ

号の読み込みを68000が失敗した場合（バスエラービンのアサート、Low→BERR）は、スプリアス割込みとして扱われ、例外ベクタ番号24が68000内で自動生成される（この場合のバスエラーは2重バス障害の対象とはならない）。このようにして生成された例外ベクタ番号を4倍して例外ベクタアドレスを生成するのが、割込みアノレジサイクルに続く2クロックのアイドル期間で行われる。

以下、68000内部状態をメモリのシステムスタック上へ退避（図）し、例外ベクタの読出し（図）を行い、新プログラムの読出しおよびプログラムカウンタのセット（図）が実行される。これで、割込み例外処理は完了し、プログラム実行状態はスーパーバイザ状態のままで、新プログラムの実行を開始する。

c. **トレース例外処理** 1つの命令を実行する毎にプロセッサの内部状態を記録する手段（トレース機能）があると、プログラムを開発するときには非常に有効である。68000



は、ステータスレジスタの T ビット（ビット番号 15）が“1”の場合、1 命令の実行が終了する毎にトレース例外処理を発生させる。この例外処理は、トレース用の各種機能をもつシステムプログラムに起動をかけることができる。T ビットが“0”の場合には、トレース例外処理は発生せず、次の命令を連続して実行する。

トレース例外処理における 68000 内レジスタとメモリとのデータのやりとりを図 8.7 に、シーケンスの詳細を図 8.10 に示す。トレース例外処理では、①スーパーバイザ状態への遷移、②例外ベクタ番号および例外ベクタアドレスの生成、③68000 内部状態のメモリへの退避、④例外ベクタの読出し、⑤新プログラムの読出しおよびプログラムカウンタのセットが順々に実行される。

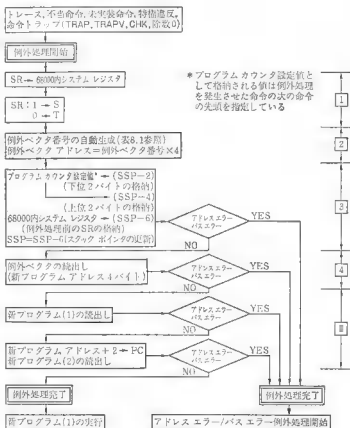


図 8.10 トレース、不当命令、未実装命令、特権違反、命令トラップの例外処理シーケンス

前述の例外処理後に実行する新プログラムで、68000 内のレジスタ等をメモリなどに出力しておけば、1 命令ごとに 68000 の動作をモニタすることができ、プログラムを開発する上で非常に助けとなる。なお、このモニタ機能を有する新プログラムは、スーパーバイザ状態で実行されるシステム プログラムの一部である。

**d. 不当命令・未実装命令例外処理** 命令として許されていないアドレス モードを指定するなど不当な命令パターンを命令として実行しようとしたときには不当命令例外処理が発生する。不当命令例外処理の詳細を図 8.7, 8.10 に示す。これは、トレース例外処理と全く同じ処理である。

68000 ではエミュレーション用の命令パターンとして、図 8.11 に示す 2 種類の未実装命令パターンが用意されている。この未実装命令を実行しようとしたときに、未実装命令例外処理が発生する。未実装命令処理の詳細を図 8.7, 8.10 に示す。例外ベクタ番号は表 8.1 で示すように、“1010” 未実装命令に対しては 10 が、“1111” 未実装命令に対しては 11 が

15	14	13	12	11															0
1	1	1	1																
無 視																			
未実装命令 1111																			
15	14	13	12	11															0
1	0	1	0																
無 視																			
未実装命令 1010																			

図 8.11 不実行命令のオペレーションワードパターン

割り当てられている。未実装命令例外処理後に実行するプログラムで、未実装命令のビットパターンを解析し、それに対応して関数演算、浮動小数点演算等の新機能をサポートした後、未実装命令に続く命令を実行するようにすれば、見掛け

上はこの未実装命令がその新機能を実行する命令となる（16 章参照）。プロセッサの動作としてみた場合には、未実装命令の実行は、ある機能をもったプログラムの起動（呼出し）操作となっている。

**e. 特権違反例外処理** 68000 ではシステムの信頼性を向上させるため、プログラム実行状態をスーパーバイザ状態とユーザ状態とに区別している（8.2 節）。このスーパーバイザ状態でだけ利用できる命令として特権命令（詳細に関しては 17 章、特権命令を参照）がある。特権違反例外処理は、この特権命令をユーザ状態で実行しようとしたときに発生する。特権違反例外処理の詳細を図 8.7, 8.10 に示す。この処理はトレース例外処理等と同様である。

**f. 命令トラップ例外処理** 命令トラップ例外処理は、TRAP 命令の実行、および TRAPV 命令、CHK 命令、除算命令の実行時にプロセッサの内部状態がある設定条件となった場合に発生する。例外処理は図 8.7, 8.10 に示すように、トレース例外処理等と同様である。

TRAP 命令を実行すると、必ず命令トラップ例外処理が発生する。例外ベクタ番号 32～47 (表 8.1 参照) は、命令のオペレーションワード下位 4 ビットで指定される。この TRAP 命令は、ユーザ状態で実行するプログラム内から、スーパーバイザ状態で実行するプログラムを呼び出す場合 (スーパーバイザコール) に利用される。

TRAPV 命令を実行すると、前の命令の実行の結果によりコンディションコードの V フラグが“1”となった場合、命令トラップ例外処理が発生する。したがって、オーバフローが発生しそうな命令の直後に TRAPV 命令を置き、例外処理後に起動される新プログラム

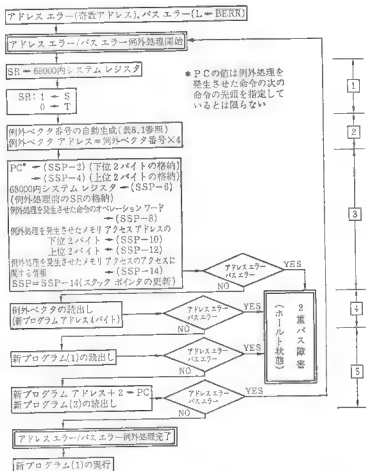


図 8.12 アドレスエラー、バスエラーの例外処理シーケンス

でその対策を行えば、オーバフローによる誤りを防ぐことができる。また、CHK 命令を実行すると、指定したデータレジスタの内容が0以下の場合およびソースオペランドで示した値よりも大きい場合、命令トラップ例外処理が発生する。詳細に関しては、16章のシステム制御命令を参照されたい。

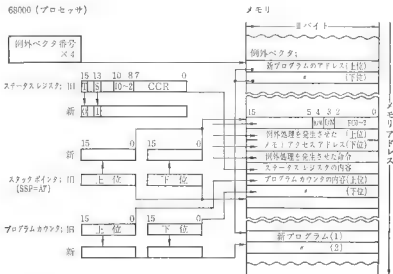
除算命令を実行し、除数が0の場合にも命令トラップ例外処理が発生する。なお、除算結果がオーバフローする場合には例外処理は発生せず、除算命令は実行されない。オーバフローを対策するためには、除算命令の後に TRAPV 命令を置き、例外処理に実行される新プログラムでエラー処理を行えばよい。除算命令に関する詳細は、13章の算術論理演算を参照されたい。

**g. アドレスエラー例外処理** ワードまたはロングワードのメモリ参照をする場合、アクセスしたメモリのアドレスが奇数であると、アドレスエラーが発生し例外処理が行われる。アドレスエラー例外処理においても、図 8.12 に示すように、①スーパーバイザ状態への遷移、②例外ベクタ番号 (=2、表 8.1 参照) および例外ベクタアドレス (=2×4=8) の生成、が実行される。レジスタとメモリの間のデータの流れを図 8.13 に示す。

アドレスエラー例外処理での特徴は、メモリ上に退避される 68000 内部状態の情報である。システムスタック上には、図 8.13 で示すように、プログラムカウンタの内容、例外処理実行前のステータスレジスタの内容のほかに、例外処理を発生させた実行中の命令のオペレーションワード、例外処理を発生させたメモリアクセスアドレス、およびそのアクセスに関する情報が退避される。メモリ上に退避されるプログラムカウンタの内容は、必ずしも命令の先頭アドレスではなく、例外処理を発生させた現在実行中の命令拡張部から次に続く命令、あるいは次に実行する予定の命令の命令拡張部までのどこか 1 つの途中のアドレスを指定している。このように、指定アドレスが一定していないので、例外処理後に実行する新プログラムのなかで、メモリ上に退避された 68000 内部状態の情報を解析して、エラー処理を行う必要がある。

68000 内部状態のメモリへの退避後には、③例外ベクタの読出し、④新プログラムの読出しおよびプログラムカウンタのセットが行われ、例外処理を完了する。

**h. バスエラー例外処理** メモリをアクセスしたとき、メモリからの応答がない場合などには、外部回路が 68000 にバスエラーの発生を知らせる必要がある。これは、バスエラー入力ピンをアサートすることにより行われる。68000 がバスエラーの発生を検出すると、バスエラー例外処理が発生する。メモリからの応答がなくても、バスエラー入力ピンのアサートが行われなければ、68000 はメモリからの応答を待ち続けることになる。



注) メモリ上に格納される 68000 内部情報

R/W: アドレスエラー、バスエラーが発生させたメモリアクセスの状態 (1: 読出し, 0: 書込み)

I/N: アドレスエラー、バスエラーが発生させたときのアクセスされたデータの内容 (1: 命令, 0: 命令以外)

図 8.13 アドレスエラー、バスエラー例外処理

バスエラー例外処理の詳細を図 8.12, 8.13 に示す。この処理はアドレスエラー例外処理と同様である。例外処理後に実行する新プログラムでエラー処理を行う場合には、アドレスエラー例外処理の項で説明したように、メモリ上に退避されたプログラムカウンタの内容値が所定のアドレスを指定していないことに注意する必要がある。

## 9

## 周辺デバイスとのインターフェイス

68000 MPU にはメモリ、68000 周辺 LSI の他に、8 ビット マイクロプロセッサ 6800 周辺 LSI を接続できる。68000 周辺 LSI については次章で述べることにし、この章では、メモリおよび 6800 周辺 LSI を接続する場合に必要なインターフェイスについて説明する。68000 MPU と周辺 LSI とを接続する場合、68000 のアーキテクチャ（2 章で説明）のうち次の 2 点が特徴となっている。

（1）メモリ マップド I/O

（2）同期および非同期バス インターフェイス

メモリ マップド I/O は、プログラムおよびデータとなる ROM、RAM のアドレスと入出力装置（I/O）のアドレスを同一アドレス空間に割り付けるものである。これによりバスを介してシステムを拡張することが容易になり、プログラムからはメモリも入出力装置も同じものとして扱うことができる。68000 バスインターフェイスは同期および非同期の両方式に適用可能であり、それぞれに必要な制御信号を有する。なお、同期バスインターフェイスは 6800 周辺 LSI と接続するためのものである。

各々の周辺デバイスとの接続を説明する前に、68000 を用いたシステムの全体構成の例を図 9.1 に示す。図 9.1 は 2 個の 68000 を用いマルチプロセッサシステムを構成した例である。68000 システム（Ⅰ）およびシステム（Ⅱ）はバスアービタ、バス結合デバイスを用いてそれぞれのローカルバスを共通バス（グローバルバス）と結合し、主メモリを共有している。さらにシステム（Ⅱ）は IPC（Intelligent Peripheral Controller）を用いてプライベートバスを設け、メモリ、I/O を拡張している。

68000 は、マルチプロセッサ構造を取りやすいような機能をもっていることも特徴の 1 つである。このように 68000 は大規模なコンピュータシステムをも実現できるが、本章ではこれらシステムを実現する上で基本となる各種周辺 LSI とのインターフェイスの例を

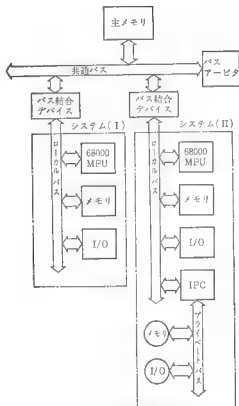


図 9.1 マルチプロセッサシステムの例

いくつかを紹介する。

## 9.1 最小システム構成

68000 のシステムを構成する場合には少なくとも次のものが必要である。

- (1) 68000 MPU
- (2) +5V 電源
- (3) TTL レベルの単相クロック
- (4) プログラムを書き込んだ ROM (または PROM) および RAM
- (5) リセット、ホールド回路

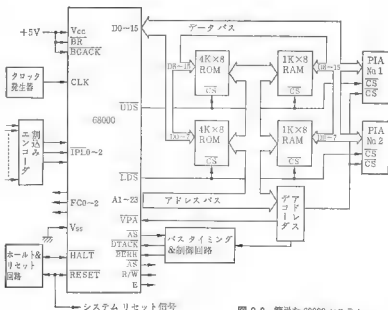


図 9.2 簡単な 68000 システム

### (6) 入出力インターフェイス用デバイス

以上の部品を使用して簡単な 68000 システムを構成した例を図 9.2 に示す。この例ではメモリとして 4K×8 ビットの ROM を 2 個使用して 4K ワードを構成し、さらに 1K×8 ビットのスタティック RAM を 2 個実装して 1K ワードを構成している。また入出力用インターフェイスとして PIA を 2 個配置している。この PIA の選択信号は  $\overline{CS}$  アドレスバスの内容をデコードすることによって得られる。

これらのメモリおよび PIA は  $\overline{UDS}$ 、 $\overline{LDS}$  によってバイト (8 ビット) 単位のアクセスも可能になっている。図では PIA No. 1 はデータワードの下位バイトを、PIA No. 2 はデータワードの上位バイトを構成するように結線されている。

バスタイミング制御回路は  $\overline{DTACK}$  信号の生成を行う。同時に、外部デバイスが割り付けられていないメモリ空間へのアクセスが生じた場合  $\overline{BERR}$  信号をアサートする。

CLK 端子にはデューティ 50% の TTL レベルのクロックを入力する必要がある。クロック発生器はこのためのものである。

割込みエンコーダはプロセッサに対しレベル 1~7 までの割込み要求を入力する。ホールド & リセット回路はリセットならびにホールドオペレーションの制御を行う。パワー



オンリセット時（電源を投入したとき）にプロセッサをリセットするためには 68000 の  $\overline{\text{RES}}$  および  $\overline{\text{HALT}}$  端子を 100ms 以上ローレベルにしなければならない。

$\overline{\text{RES}}$  および  $\overline{\text{HALT}}$  端子はオープン ドレイン端子であり、このため TTL の外部回路ではオープン コレクタ形式の素子を使用しなければならない。また、必要に応じて、3章の図 3.14 で示した簡易シングルステップ回路を付加することもある。

## 9.2 ROM およびスタティック RAM とのインターフェイス

68000 は 23 本のアドレス線 A1~A23 をもっており、 $2^{23}$  ワードのアドレスを指定することができる。また、3章で述べたファンクションコード (FC0~FC2) を使用すれば、さらにアドレス空間を 4 倍に広げることが可能である。上位データ ストローブ ( $\overline{\text{UDS}}$ ) および下位データ ストローブ ( $\overline{\text{LDS}}$ ) によりバイト毎のアクセスもできる。図 9.3 に ROM, スタティック RAM とのインターフェイス例を示す。この例ではすべての

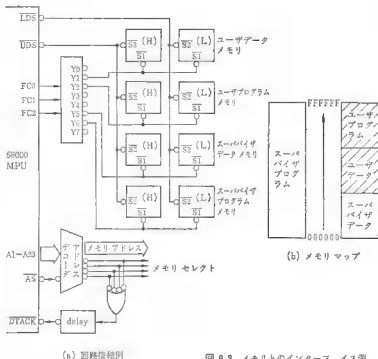


図 9.3 メモリとのインターフェイス例

メモリに FC0~FC2 を与えることにより、メモリ保護を行っている。また  $\overline{UDS}$ ,  $\overline{LDS}$  によりバイトアクセスが可能である。プロセッサの2クロックサイクル以内でアクセスできない速度の遅いメモリを使用する場合には、メモリセレクト信号が発生してからメモリが完全に応答するまでの時間を遅延回路を介して  $\overline{DTACK}$  に入力する必要がある。プロセッサの2クロックサイクル以内で十分アクセスできる高速のメモリを使用する場合にはこの遅延回路は不要である。

68000 は例外ベクタとしてスーパーバイザプログラム領域の 0~3 番地およびスーパーバイザデータ領域の 4~255 番地までを予約している。したがってメモリを割り付ける場合にはこの点に注意しなければならない。スーパーバイザデータ領域の 256 番地~1023 番地までは、ユーザからの割込みによる例外ベクタの領域として使用する。この領域を単なるデータ領域として使うことはさけたほうがよい。

### 9.3 ダイナミック RAM とのインターフェイス

大容量の RAM をもつシステムを構成する場合は、高集積が可能なダイナミック RAM を使用すると RAM の使用個数が減りコストの点で有利である。ダイナミック RAM は記憶情報を保持するために、適当な周期でリフレッシュ動作を行わなければならない。通常のダイナミック RAM では、この周期は約 2ms である。

たとえば、日立 HM4816 のような 16K ビットのダイナミック RAM では、128 行×128 列のメモリ構成となっており、アドレスバスの 7 本で行アドレスを与え、別の 7 本で

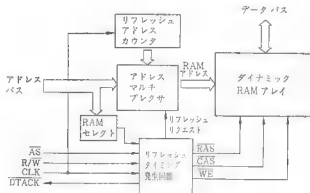


図 9.4 ダイナミック RAM 駆動回路ブロック図

列アドレスを与えることになる。つまり 2ms 間に行数 (128) 回だけのリフレッシュサイクルが必要である。

図 9.4 にダイナミック RAM の駆動回路のブロック図を示す。リフレッシュアドレスカウンタはリフレッシュサイクル中に RAM のリフレッシュアドレスを発生させるバイナリカウンタである。上記の 16K ビットのダイナミック RAM の場合は 7 ビットのカウンタが必要である。カウンタアップの入力はシステムクロックを分周して適当な周期を得る。

アドレスマルチプレクサはリフレッシュアドレスとシステムアドレスバスの内容を切り換えて RAM のアドレスを発生させるものである。

アドレスマルチプレクサはリフレッシュサイクル中にはリフレッシュアドレスを選択する。プロセッサからのメモリアクセスとリフレッシュサイクルが統合した場合、リフレッシュサイクルが優先される。リフレッシュタイミング発生回路はリフレッシュサイクルが終了するまでの間  $\overline{\text{DTACK}}$  をアサートしないことにより、プロセッサのメモリアクセスを待たせるわけである。つまり、この場合プロセッサからのメモリアクセス時間を見掛け上引き延ばされたようになる。

## 9.4 割込みベクタ発生回路

割込みにはオートベクタ割込みおよびベクタ割込みがある。この両者の識別は VPA 入力で行われ、68000 が割込み要求を受け付けたことを示す割込みアクノレージサイクルで  $\overline{\text{VPA}}$  が外部からアサートされた場合、68000 はオートベクタ割込みとみなす。オートベクタ割込みが要求されると、プロセッサは所定の割込みベクタを内部で発生する。一方、外部からベクタ割込みを要求する際にはデータバス D0~D7 を介して例外ベクタ番号を入力する。図 9.5 に割込み回路の例を示す。この例ではレベル 3 の割込み (INT3) がオートベクタ割込み、レベル 5 の割込み (INT5) がベクタ割込みになっている。MPU が割込みを受け付けるとアドレスバス A1~A3 から割込みレベルを送出するとともに FC0~FC2 をすべて "High" にする。 $\overline{\text{VPA}}$  入力は 6800 周辺 LSI とのインターフェイスのためのコントロール信号にも使用されるため、ワイヤード OR 論理が可能なオープンコレクタ形式の素子を外部回路に用いる必要がある。

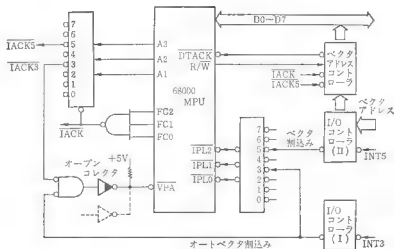


図 9.5 割込み回路の例

## 9.5 6800 周辺 LSI とのインターフェイス

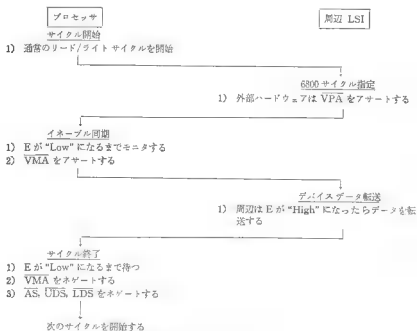
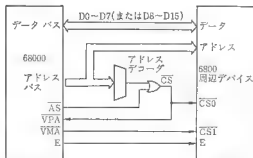
68000 MPU は 6800 周辺 LSI と直接バス コンパチブルである。6800 周辺 LSI の主なものを以下に示す。

- 6821 周辺インターフェイス アダプタ (PIA)
- 6843 フロッピディスク コントローラ (FDC)
- 6845 CRT コントローラ (CRTC)
- 6850 非同期コミュニケーションインターフェイス アダプタ (ACIA)
- 6852 同期シリアルデータ アダプタ (SSDA)

これらの 6800 周辺 LSI は同期式デバイスである。68000 とインターフェイスをとるためのブロック図を図 9.6 に示す。アドレス デコーダとアドレス ストローブ信号 (AS) により 6800 周辺 LSI のアドレスが検出されると、68000 MPU は 6800 のバス タイミング条件を満足するようにバス サイクルを修正する。このインターフェイスのために次の 3 つの信号が使用される。

- E (イネーブル)
- $\overline{VMA}$  (バリッド メモリ アドレス)
- $\overline{VPA}$  (バリッド ペリフェラル アドレス)

図 9.6 6800 周辺 LSI (デバイス) とのインターフェイス



■ 9.7 6800 周辺 LSI とのインターフェイスフローチャート

図 9.7 に 6800 周辺 LSI とのインターフェイス操作のフローチャートを示す。E 信号は 68000 の入力クロック周波数の 1/10 の一定周波数をもつクロックである。これは 6800 システムにおける E または  $\phi_2$  に相当するバス クロックである。

VPA はアドレスされたデバイスが 6800 周辺 LSI であり、データ転送が E 信号と同期して行われるべきであることを 68000 MPU に知らせる。また VMA はアドレスバス上に

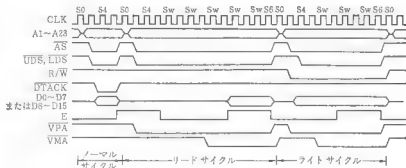


図 9.8 6800 周辺 LSI とのタイミング

有効なアドレスがあり、プロセッサが信号と同期して動作していることを示し、 $\overline{VPA}$  入力にのみ応答する。 $\overline{VMA}$  信号は 6800 周辺 LSI のチップセレクト条件の一部として使用される。

以下図 9.8 を用い 6800 周辺 LSI とのインターフェース タイミングを説明する。

(1) ステート 0 (S0) のサイクルのときアドレスバスとファンクションコード (F0~F2) はハイインピーダンス状態となり、半クロック後の S1 でアドレスバスとファンクションコードの出力はハイインピーダンス状態から開放される。

(2) S2 において  $\overline{AS}$  がアサートされ、アドレスバス上に有効アドレスがあることを示す。

バスサイクルがリードサイクルの場合には  $\overline{UDS}$ ,  $\overline{LDS}$  とも S2 でアサートされる。バスサイクルがライトサイクルの場合には  $R/\overline{W}$  信号は S2 でライト ("Low") に切り換え、半クロック後の S3 でライトデータをデータバス上へ送り出し、S4 でデータストローブを出力する。

$\overline{VPA}$  入力信号は  $\overline{AS}$  がアサートされているときに、アドレスバスを外部回路でデコードすることによって作られる。

(3)  $\overline{VPA}$  検出後、プロセッサは E がネゲート ("Low") されるまで、必要に応じて待ち状態 Sw に入り、その後  $\overline{VMA}$  をアサートする。周辺 LSI は E 信号が "High" の間に処理を実行する。

(4) リードサイクルのときには S6 で周辺 LSI からのデータをラッチする。リードサイクル、ライトサイクルのどちらにおいても、プロセッサは S7 で  $\overline{AS}$  とデータストローブをネゲートする。信号はこのときに "Low" になる。

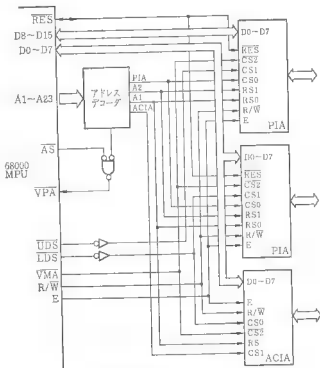


図 9.9 6800 周辺 LSI とのインターフェイス例

(5) 周辺回路は  $\overline{AS}$  がネゲートされた後、1クロック以内に  $\overline{VPA}$  をネゲートしなければならない。

図9.9に68000と6821 PIA, 6850 ACIA とのインターフェイスの例を示す。2個のPIAはワード(16ビット)毎の入出力とバイト(上位8ビット, または下位8ビット)単位の入出力が可能になっており, ACIAはシステムバスの下位8ビットに接続されている。ここではPIA, ACIAの簡単なインターフェイス例について示した。その他の6800周辺LSIを用いてシステムを拡張することもできる。

# 10

## 68000 ファミリ周辺 LSI

マイクロコンピュータを種々の応用システムに適用するとき、MPU 自身の性能とともにその周辺 LSI がいかに豊富であるかがそのシステムの性能やシステム開発にとって重要な鍵となる。このため 68000 でも MPU をサポートする周辺 LSI の充実が図られており、現在までに表 10.1 に示すような周辺 LSI が提供されている（開発中のものも含む）。

表 10.1 68000 ファミリ周辺 LSI

品番号	略 称	名 称
68450	DMAC*	Direct Memory Access Controller
68451	MMU*	Memory Management Unit
68120	IPC*	Intelligent Peripheral Controller
68230	PI/T*	Parallel Interface/Timer
68122	CTC	Cluster Terminal Controller
68452	BAM	Bus Arbitration Module
68590	LANCE**	Local Area Network Controller
68852	MPCC	Multi-Protocol Communications Controller
68653	PGC	Polynomial Generator Checker
68661	EPCI	Enhanced Programmable Communication Interface
68881	FPCP**	Floating Point Co-Processor

\* 本章で説明する、\*\* 開発中

この章では、システムを構成する上で特に重要な 68450 DMAC、68451 MMU、68120 IPC、68230 PI/T について説明を行う。

### 10.1 68450 DMAC (Direct Memory Access Controller)

#### 10.1.1 68450 DMAC の特徴

68450 DMAC (Direct Memory Access Controller) は DMA 転送を制御する 68000 周



辺 LSI である。DMA 転送とは、図 10.1 に示すとおり、入出力装置とメモリあるいはメモリとメモリとの間のデータ転送を直接 (MPU を介さず) 行う機能である。図 10.1 において MPU はシステムバスから切り離された状態にあり、DMAC がバス マスタ

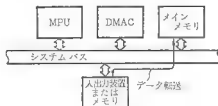


図 10.1 DMA 転送

権をもちデータの転送を制御する。この制御により大量のデータ転送を高速に行うことができる。DMAC はコンピュータ システムでは最も重要な周辺デバイスの 1 つである。

68450 DMAC は 64 ピン パッケージに実装されており以下の機能をもっている。

- (1) 68000 MPU および 6800 ファミリとバス コンパチブルである。
- (2) 独立した 4 個のチャネルをもつ。
- (3) 転送速度は最高 4 M バイト/秒まで可能である。
- (4) 論理データ幅は 8, 16, 32 ビットである。
- (5) 物理バス幅は 8, 16 ビットである。
- (6) 転送リクエス ト モードとして次の 3 とおりが用意されている。
  - i) サイクルスチールモード
  - ii) バーストモード
  - iii) オートリクエス ト モード (メモリ-メモリ間転送)
- (7) 転送アドレッシングとして次の 2 とおりが可能である。
  - i) シングルアドレッシングモード
  - ii) デュアルアドレッシングモード
- (8) 次の 3 とおりの方法によりマルチブロック転送ができる。
  - i) 継続動作法
  - ii) アレイチェイニング
  - iii) リンクチェイニング

項目 (6), (7), (8) については 10.1.3 項にて説明する。

### 10.1.2 信号線とバス サイクル

図 10.2 に 68450 DMAC の入出力信号の構成を示す。また、表 10.2 に各信号線の機能をまとめる。

表 10.2 68450 DMAC の信号線

略 号	信 号 名	機 能
A8~A23/D0~D15 CS	— Chip Select	アドレス線 A8~A23/データ線 D0~D15 チップが選択されたことを示す入力信号
A1~A7 AS, LDS, UDS R/W, DTACK, BR, BG, BGACK Vcc, Vss, CLK	—	68000 MPU と同じ機能 (3 章参照)
IRQ	Interrupt Request	MPU に対する割込み要求出力
IACK	Interrupt Acknowledge	MPU が割込みを受け付けたことを示す入力信号
OWN	Own	DMAC がバス マスタであることを示す
UAS	Upper Address Strobe	A8~A23/D0~D15 のうち A8~A23 が有効であることを示す
HIBYTE	High Byte	上位バイトが有効 (バイト転送時のみ使用) であることを示す
DBEN	Data Bus Enable	データバスが有効である
DDIR	Data Direction	データ転送の方向を示す
BEC0,1,2	Bus Exception Controls	BEC0,1,2= 000 リセット, 100 未定義 001 リトライ, 101 バスエラー 010 未定義, 110 バス解放後リトライ 011 ホールト, 111 例外条件なし
FC0,1,2 REQ0,1,2,3 ACK0,1,2,3 PCL0,1,2,3 DONE DTC	Function Codes Requests Acknowledges Peripheral Control Lines Done Device Transfer Complete	FC0,1,2=111 は未定義, 他は 68000 と同じ DMA 転送要求 DMAC が DMA 転送要求を受け付けた ことを示す 入出力装置制御線 (詳細省略) ブロック転送が終了したことを示す 1 回のデータ転送が完了したことを示す

図 10.4 は 68450 DMAC の制御の下で、メモリから ACK ピンを有するデバイスへ 16 ビットのデータを転送する際のフローチャートである。これは DMA 転送機能のうちで最も基本的なものである。68000 のワードリードフローチャート (3 章、図 3.4 参照) と比較すると次の相違点があるが、基本となる転送手順は同じである。

- (1) DMAC が  $\overline{\text{ACK}}$  のアサートおよびネゲートを行う。
- (2)  $\overline{\text{ACK}}$  でアサートされたデバイスがデータを取り込む。
- (3)  $\overline{\text{DTC}}$  信号により転送の終了を知らせる。

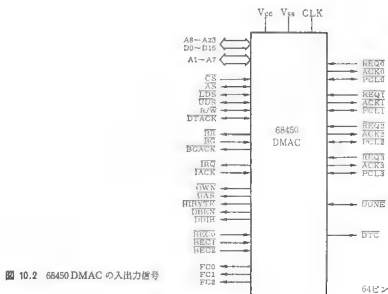
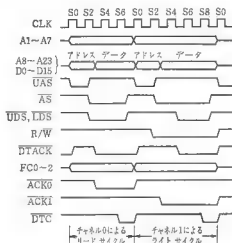


図 10.2 68450 DMAC の入出力番号



■ 10.3 68450 DMAC のリードサイクルおよびライトサイクルのタイミング

図 10.3 は図 10.4 で示す DMA 転送フローをタイミングチャートで表したものである。リードサイクル（メモリ→I/O デバイス）は 4 クロックで、ライトサイクル（I/O デバイス→メモリ）は 5 クロックで完了する。68450 は独立した 4 個のチャネルをもっておりバス権を保有したまま別のチャネルのデータ転送を行うこともできる。この■、チャネル切換

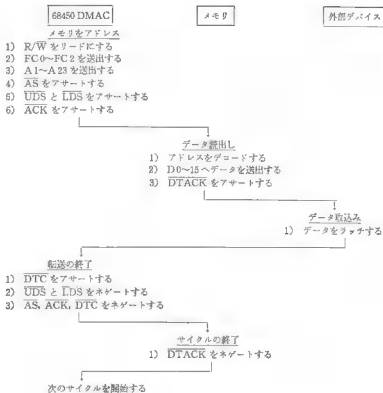


図 10.4 68450 DMAC によるワードリード サイクルのフローチャート

えによるオーバーヘッドはない。なお、リード/ライトの定義はメモリを中心としている。

### 10.1.3 転送プロトコル

本項では 68450 DMAC の転送プロトコル、すなわち 10.1.1 項であげた転送リクエストモード、転送アドレッシングおよびマルチブロック転送について述べる。

**a. 転送リクエストモード** 転送リクエストモードとは DMA 転送の要求方法のことであり、68450 DMAC では 10.1.1 項でも述べたように、サイクルスチールモード、バーストモード、オートリクエストモードの 3 とおりがある。

サイクルスチールモードでは、リクエスト信号  $\overline{\text{REQ}}$  がアサートされたときに 1 回の DMA データ転送が行われる。すなわち、このモードでは DMA 転送は MPU のバスサイクルの間に 1 バスサイクルだけ割り込んで行われる。したがって、比較的低速のデー

タ転送に用いられる。バーストモードは、 $\overline{\text{REQ}}$  信号がアサートされたときに1ブロックのデータ転送が行われるものである。入出力装置はDMACからのアクノレッジ信号  $\overline{\text{ACK}}$  によりデータの入出力を行う。1ブロックの転送終了は終了信号  $\overline{\text{DONE}}$  により行われる。このモードでは1ブロックの転送の間、DMACがバスを占有できるので高速転送に用いられる。オートリクエストモードは、一種のバーストモードであり内部的に  $\overline{\text{REQ}}$  信号が生成される。メモリは  $\overline{\text{REQ}}$  信号を生成する機能をもたないので、メモリ-メモリ転送はオートリクエストモードを使う必要がある。転送速度などはDMACの内部レジスタの値を変えることにより変更できる。以上3種類の転送リクエストモードをまとめると表10.3のとおりとなる。

表 10.3 68450 DMAC における転送リクエストモードの比較

転送リクエストモード	転送速度	転送デバイス
サイクルスチール	低速	入出力装置-メモリ
バースト	高速	〃
オートリクエスト	高速/低速	メモリ-メモリ

**b. 転送アドレッシング** 68450 DMAC の転送アドレッシングには10.1.1項でも述べたようにシングル/デュアルアドレッシングの2種類がある。これらの相違点はDMACがアドレス線をとおして選択すべき外部デバイスの個数が1あるいは2個というところにある。シングルアドレッシングでは、DMACは $\overline{\text{ACK}}$ 信号ならびにアドレス線A1~A23、バス制御信号で送信デバイスと受信デバイスを選択する。一方、デュアルアドレッシングでは送信デバイスと受信デバイスの選択はアドレス線をとおし2回に分けて行われる。デュアルアドレッシングでは転送すべきデータを一度DMAC内部のホールディングレジスタに格納した後、受信デバイスへの書き込み操作を行う。したがって、1回の転送でリードおよびライトの両サイクル（合計9クロック）が必要となる。 $\overline{\text{ACK}}$ 信号をもたないメモリ-メモリ間転送に対して有効である。また、デュアルアドレッシングでは、2個の8ビットデータを16ビットデータにパックした後転送するデータパックング機能がある。以上の転送アドレッシングについて表10.4にまとめる。

表 10.4 68450 DMAC における転送アドレッシング法の比較

転送アドレッシング	クロック数/1転送	転送デバイス	パックング機能
シングル	4クロック（リード） 5クロック（ライト）	入出力装置-メモリ	なし
デュアル	3クロック	メモリ-メモリ	あり

図 10.5 マルチブロック転送

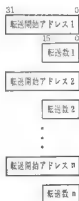


図 10.6 アレイチェイニングによるマルチブロック転送指定フォーマット

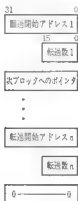
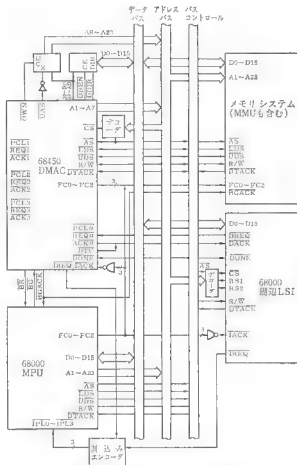


図 10.7 リンクチェイニングによるマルチブロック転送指定フォーマット

**c. マルチブロック転送** マルチブロック転送とは図 10.5 に示すように複数のブロック（一般にブロック長は異なる）を転送するものである。68450 DMAC では 10.1.1 項でも述べたように、継続動作法、アレイチェイニング、リンクチェイニングによってマルチブロック転送ができる。継続動作法では原則として転送開始アドレスおよび転送数を毎回設定する必要がある。しかし直前のブロック転送終了時の値がレジスタに保持されているので、同じブロック長でかつ引き続いた番地ヘデータを転送する場合には継続動作の指令のみをすればよい。アレイチェイニングにおいては図 10.6 で示すように転送開始アドレスおよび転送数をメモリ上にアレイ形式で設定する。リンクチェイニングでは図 10.7 で示すように転送開始アドレスおよび転送数をリスト形式でメモリ上に指定する。

#### 10.1.4 68450 DMAC を使用したシステム構成例

図 10.8 に、68450 DMAC、68000 MPU、メモリシステム(MMUを含む)および68000 周辺 LSI を用いて、システムを構成した場合の接続参考図を示す。プルアップ抵抗等の詳細については省略してある。



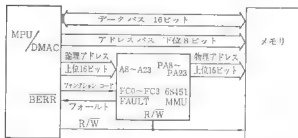


図 10.9 68451 MMU を用いた論理-物理アドレス交換システムのブロック図

レスを出力として生成する。MMU は書き込み違反ならびに未定義セグメント（論理アドレスに対応する物理アドレスが未定義）を検出したならば、フォールト信号をバス マスタへ返す。論理アドレス線の下位 8 ビットはアドレス変換されずセグメント内のアドレスを示す。すなわち 1 つのセグメントの大きさは最小 256 バイトである。

このように 68451 MMU を用いてシステムを構成すれば、より厳重なメモリ管理を効率良く（ソフトウェアでするよりも）実現でき、OS の負担が軽くなる。しかし、このアドレス変換によって生ずるオーバーヘッド（次項で記述）は必ずしも無視できない場合もあり得る。したがって、68451 MMU は小さなシステムよりも大きなシステムに適したものであるといえる。

68451 MMU は次の特徴をもつ。

- (1) 68000 MPU および 68450 DMAC とコンパチブル
- (2) 24 ビットアドレスバスのうち上位 16 ビット (A8~A23) をアドレス変換する。
- (3) 書き込みチェックを行う。
- (4) 1 つの MMU で一冊に 32 個までのセグメントに対しアドレス変換ができる。
- (5) マルチ MMU システムへ容易に拡張できる。
- (6) 厳重なメモリ管理が実現でき、OS の負担を軽減できる。

### 10.2.2 信号線とアドレス交換のバス サイクル

図 10.10 に 68451 MMU の入力出力信号の構成を示す。64 ピンのパッケージに実装されている。表 10.5 に各信号線の機能をまとめる。

図 10.11 にアドレス変換時のバス サイクル タイミング チャートを示す。基本的には 68000 MPU のリード/ライト サイクルと同じである。相違点は、MMU が  $\overline{AS}$  信号を受



図 10.10 68451 MMU の入出力信号

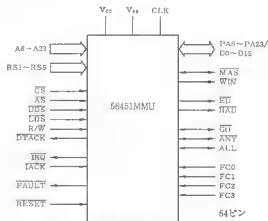


表 10.6 68451 MMU の信号線

略 号	信号名	機 能
A8~A23, AS, UDS, LDS, R/W, DTACK, Vcc, GND, CLOCK, RESET	—	68000 MPU と同じ機能 (3章参照)
CS, IRQ, LACK, FC0, 1, 2	—	68450 DMAC と同じ機能 (前節参照)
RS1~RS5 FAULT	Register Selects Fault	内部レジスタ選択線 書込み違反, 未定義セグメントを検出したことを示す
PA8~PA23/D0~D15 MAS WIN	Physical Address/Data Mapped Address Strobe Write Inhibit	物理アドレス線/データ線 物理アドレス線が有効であることを示す 書込み禁止を表す(リード/モディファイライト時に使用)
ED	Enable Data	PA8~PA23/D0~D15 のデマルチプレクシングに使用する
HAD	Hold Address	
GO, ANY, ALL	Global Operation Any, All	マルチ MMU システムを構成するときに使用する
FC3	Function Code 3	BACK 信号を入力する

け取ってからアドレス変換を開始し、物理アドレス ストローブ信号  $\overline{\text{MAS}}$  でメモリのアクセスが行われることである。図中 "Sw" は 68000 MPU の持ちサイクルを表し。リードサイクルで2クロック、ライトサイクルで1クロックがアドレス変換のオーバーヘッドとな

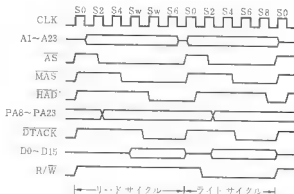


図 10.11 68451 MMU を用いた 68000 システムのリード サイクルおよびライトサイクルのタイミング

る。ライトサイクルにおいて待ちサイクルがない理由は次のとおりである。図 10.11 に示すように、リードサイクルでは 68000 MPU は  $\overline{\text{MAS}}$  信号が確定した後メモリからデータを読み出すのに対し、ライトサイクルでは  $\overline{\text{AS}}$  信号受信と同時にメモリデータを送出できるからである。

### 10.2.3 アドレス変換

本項では 68451 MMU におけるアドレス変換の方式について述べる。68451 MMU では図 10.12 で示すように“デスクリプタ”とよばれるレジスタブロックが 32 個用意されている。デスクリプタは 3 個のレジスタ (計 9 バイト) からなり、アドレス変換機能の中心となるものである。デスクリプタ中の各レジスタの機能についてはアドレス変換過程の説明において適宜ふれる。68451 MMU のアドレス変換は次の 3 段階に分けることができる。

- (1) デスクリプタの選出
- (2) 書込みチェック
- (3) 物理アドレスの生成

以下に各段階の機能について述べる。

a. デスクリプタの選出 アドレス変換を行うにあたり、68451 MMU は 32 個のデ

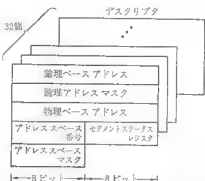


図 10.12 68451 MMU のアドレス交換用デスクリプタ

スクリプタのうちどれを使うのか決定する必要がある。このために4本のファンクション<sub>R</sub>ノコード (FC0~FC3) および論理アドレス線 (A8~A23) を入力として使用する FC3 には  $\overline{\text{BGACK}}$  を入力する。まずファンクション<sub>R</sub>ノコードから内部的にアドレススペース番号という情報を生成し、同じアドレススペース番号をもつデスクリプタを選び出す。この選出は連想記憶方式によって行われる。該当するデスクリプタが存在しない場合  $\overline{\text{FAULT}}$  信号を生成し、アドレス変換を中断する。次に、選択したデスクリプタの論理ベースアドレスと入力アドレス線が一致するかどうかに判定する。一致しない場合は  $\overline{\text{FAULT}}$  信号を生成しアドレス変換を中断する。

アドレススペース番号の一致判定および論理アドレスの一致判定においては、それぞれマスク用レジスタ (アドレススペースマスク、論理アドレスマスク) が用意されている。後者を例にとり図 10.13 を用いマスク機能について説明する。図で示すように論理アドレスマスクの内容が“0”であるビット位置が一致判定の対象外となる。たとえば A8~A11 の4ビットがマスクされた場合、これらのアドレス線は一致判定の対象外となるため実質的にセグメントの大きさが2の11乗バイト、すなわち2048バイトまで広がったことになる。マスクの効果についてはアドレススペース番号の一致判定も同じである。



図 10.13 論理アドレス (入力) と論理ベースアドレス (デスクリプタ) との一致判定

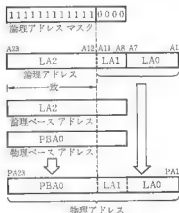


図 10.14 物理アドレスの生成例

**b. 書き込みチェック** デスクリプタのセグメント状態レジスタにはそのセグメントが書き込み可能かどうかを示すビットがある。書き込み禁止のセグメントに対し、R/W 信号が書き込みを示すとき  $\overline{\text{FAULT}}$  信号を生成する。このほかセグメント状態レジスタには、セグメントに対するアクセスの有無、書換えの有無などの情報を示すビットがある。これらはセグメントのスワッピングをはじめとする OS 機能のサポートに有用である。

**c. 物理アドレス生成** 前記 (1)、(2) の処理を受けたセグメントに対して物理アド

レスが与えられる。図 10.14 に物理アドレスの生成例を示す。アドレス線 A1~A7 ならびに論理アドレスマスクでマスクされたアドレス線はそのまま物理アドレスとなる。マスクされなかった部分は物理ベースアドレスで置き換えられる。

### 10.3 68120 IPC (Intelligent Peripheral Controller)

68120 IPC は入出力装置を制御する汎用のプロセッサであり、MPU と被制御入出力装置との間に接続される。68120 IPC を用いたシステム構成例を図 10.15 に示す。システム

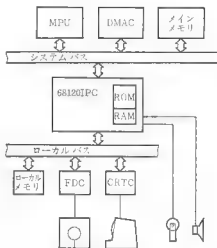


図 10.15 68120 IPC を用いたシステム構成例

バスには MPU, DMAC, メインメモリ, IPC が接続されており、ローカルバスにはローカルメモリ, FDC (Floppy Disk Controller), CRTC (CRT Controller) が接続されている。また IPC はランプ、ブザー等簡単な入出力装置を直接制御することができる。図 10.15 において 68120 IPC の内蔵 ROM あるいはローカルメモリにプログラムを格納することにより 68120 IPC にインテリジェンスをもたせることができる。したがって、MPU は入出力機能のかんがりの部分を IPC に肩代りさせることができ、それだけ MPU の

負荷が軽減されシステムパフォーマンスの向上につながる。

68120 IPC は以下の特徴をもつ。

- (1) 68000 MPU および 68000 ファミリーとバスコンパチブルである。
- (2) 8 ビットプロセッサ 6801 を核にしたアーキテクチャをもつ。
- (3) 2048 バイトの ROM を内蔵している。
- (4) 128 バイトのデュアルポート RAM を内蔵している。
- (5) タスク同期用レジスタ (セマフォアレジスタ) を 6 個もっている。
- (6) 内蔵の ROM, RAM の拡張に関して次の 3 とおりの動作モードが可能である。

i) シングルチップモード

ii) 拡張形非マルチプレックス モード

iii) 拡張形マルチプレックス モード

(7) タイマ機能を有する

(8) シリアル通信機能を有する。

(9) ROM を内蔵しない 68121 というバージョンもある。

図 10.16 に 68120 IPC の入出力信号の構成を示す。48 ピンのパッケージに実装されている。図 10.16 において図中左側のシステム アドレス線から  $\overline{\text{RESET}}$  までは 68000 インターフェイス用である。一方、図中右側の信号線は 68120 IPC が制御する入出力装置とのインターフェイス用である。このうちポート 3、SC1、SC2、ポート 4 は 68120 IPC の動作モードにより機能が変わる。これらの機能の相違を表 10.6 に示す。表 10.6 には各動作モードにおける入出力ポートの総数ならびにローカルメモリ空間の大きさも示してある。シングルチップモードでは計 21 本の入出力ポートが利用でき、拡張非マルチプレックス、拡張マルチプレックスモードではローカルメモリ空間上に入出力コントローラあ

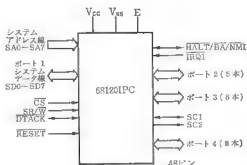


図 10.16 68120 IPC の入出力信号

表 10.6 68120 IPC の動作モードと信号線の機能

動作モード	シングルチップ	拡張非マルチプレックス	拡張マルチプレックス
信号線			
ポート 2 (5本)	入出力ポート (パラレルポート, シリアルポート, タイマ)		
ポート 3 (8本)	入出力ポート	D0~D7	A0/D0~A7/D7
SC1 (1本)	$\overline{\text{OS}}^{*1}$	R/ $\overline{\text{W}}$	R/ $\overline{\text{W}}$
SC2 (1本)	$\overline{\text{IS}}^{*2}$	$\overline{\text{IOS}}^{*1}$	AS
ポート 4 (8本)	入出力ポート	A0~A7	A8~A15
入出力ポート総数(含ポート 2)	21 本	11 本	5 本
ローカルメモリ空間	—	128 バイト	64 K バイト

\*1 Output Strobe, \*2 Input Strobe, \*\* I/O Strobe

るいはローカルメモリの割当てができる。

ポート 2 は入出力ポートとしても使えるが、内部レジスタを書き換えることによりタイマ/シリアル通信インターフェイスとしての機能をもつ。タイマ機能として次の 3 つの条件のときに内部割込みを生成し得る。各々の割込みはマスクをすることもできる。

- (1) フリーランカウンタ (16 ビット) が、内部レジスタに設定した数値と一致した場合
- (2) フリーランカウンタがオーバフローした場合
- (3) ポート 2 の最下位ビット (ビット 0) のレベル変化 (0→1, 1→0, 指定可能) を検出した場合

シリアル通信インターフェイスとしては、1 個の全 2 重 (full-duplex) 形非同期チャネルをもっている。転送レートおよび信号形式 (Non Return to Zero または Bi-phase) はプログラム可能である。68120 IPC には、パケット通信をサポートするためにウエイタップという機能がある。これは、MPU が不要パケット (アドレスが一致しない) であると判断した時点で IPC に対し不要パケットの転送を停止させ、そして次のパケットから再び IPC がデータ転送を行うものである。この機能を使うと、MPU はパケットのアドレス部を調べデータ部の取捨を決めればよく、不要なデータ転送がなくなる。

68120 IPC は 128 バイトのデュアルポート RAM を内蔵している。この RAM は 68120 IPC の内部プロセッサおよびメインシステムの両方から直接アクセスすることができ、IPC とメインシステムとのデータ転送に使う。しかし、IPC の内部プロセッサとメインシステムが同時に、同じアドレスをアクセスすることは避けなければならない。このため、68120 IPC には 6 個のセマフォアレジスタが用意されている。図 10.17 にセマフォアレジスタの構成を示す。ビット 7 の SEM (Semaphore) ビットはテストアンドセットのビット<sup>†</sup>である。ビット 6 の OWN (Ownership) ビットは、SEM ビットをセットしたプロセッサが IPC 自身がメインシステムであるかを示すためのものである。

7	6	5	4	3	2	1	0
SEM	OWN	0	0	0	0	0	0

図 10.17 68120 IPC のセマフォアレジスタ

<sup>†</sup> 3 章および 13 章参照

## 10.4 68230 PI/T (Parallel Interface/Timer)

68230 PI/T は入出力ポート、ハンドシェイク機能に加えタイマ機能を有する LSI であり入出力装置の制御用として使う。68230 PI/T は以下の特徴をもつ。

- (1) 68000 とバス コンパチブルである。
- (2) 最大 24 本までの入出力ポートをもつ。
- (3) 24 ビット カウンタと 5 ビット プリスケアラを内蔵し計時タイマとして使用できる。
- (4) 図 10.18 で示すように次の 3 おりのモードで外部へタイミング信号を発生することができる。
  - i) パルス波形モード (同期, デューティ可変)
  - ii) 矩形波形モード (同期のみ可変)
  - iii) ワンショットモード

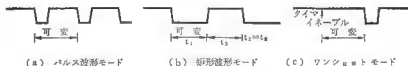


図 10.18 68230 PI/T におけるタイミング発生機能

図 10.19 に 68230 PI/T の入出力信号の構成を示す。48 ピンのパッケージに実装されている。RS1~RS5 (Register Select) は PI/T の内部レジスタを選択するための信号線で

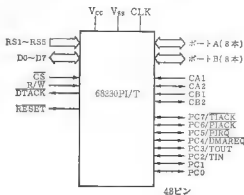


図 10.19 68230 PI/T の入出力信号

ある。データバス D0~D7 が 8 ビット幅なので 68000 MPU の MOVEP 命令を使うのが適している。ポート A、ポート B はそれぞれ 8 ビット幅であり内部は 2 重バッファ構成となっている。これらのポートをまとめて 16 ビット幅のポートとして使うことも可能である。CA1, CA2 (Strobe Control) はポート A のハンドシェーク用信号線であり、ラッチのタイミングを指定したり、バッファの空き状態の表示などに使われる。CB1, CB2 はポート B のハンドシェーク用信号線またはポート A, B を 16 ビット幅として使うときのハンドシェーク用信号線である。

PC0~PC7 は 8 ビット幅のポートとして使うこともできるが PC2~PC7 は内部レジスタを書き換えれば割込み制御、タイマ制御などにも使用できる。PIRQ (Port Interrupt Request), PIACK (Port Interrupt Acknowledge) はポート割込み制御用の信号である。TIACK (Timer Interrupt Acknowledge), TOUT (Timer Out), TIN (Timer In) は 68230 PI/T のタイマ機能に関するものである。68230 PI/T は前述したとおり 24 ビットのカウンタ、5 ビットのプリスケイラを内蔵している。基本クロックとして内部クロックまたは TIN からの外部クロックを使用できる。TOUT からは図 10.18 で示した 3 つのモードで外部へ信号を出力できる。TOUT をシステムの割込み要求線へ接続し、TIACK により割込みベクトルの発生が可能となる。

68230 PI/T のタイマ機能は OS が計時用として使用したり、タスクの切換え用として使用するのに適している。



## Ⅱ 68000 のソフトウェア

# 11      ア   セ   ン   プ   ラ

この章では 68000 システムの典型的なアセンブラ<sup>†</sup>の使用法、このうち特にソースプログラムの記述法について説明する。

アセンブラは、ユーザが記述したソースプログラムを機械語に変換するための言語処理プログラムである。ユーザはソースプログラムを、アセンブリ言語すなわち機械命令や制御命令などを表すニモニック記号を用いて記述する。このソースプログラムは図 11.1 に示すようにアセンブラの入力となる。アセンブラで処理されたプログラムはオブジェクトモジュールとなる。これは仮のアドレス付けをされた機械語の形式になっている。オブジェクトモジュールはリンケージエディタとよばれる結合編集プログラムによって編集され、実際のアドレスを割り当てられる。こうしてプロセッサが実行可能なロードモジュールが作り出される。

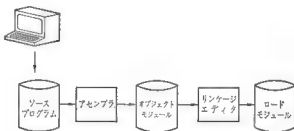


図 11.1 ソースプログラムの変換

<sup>†</sup> 一般にアセンブラは使用するシステムによって異なるのがふつうである。本書では最も標準的な 68000 アセンブラを想定している。

## 11.1 ソースプログラムの構造

ソースプログラムは1つ以上のステートメントによって構成される。ステートメントとは、アセンブリ言語で書かれた1行の文のことで、1行の有効な範囲は1カラムから72カラムまでである。

ステートメントには実行命令、アセンブラ制御命令、コメント文の3種類がある。

実行命令は機械語と1対1に対応するステートメントであり、ソースプログラムの中心をなすものである。アセンブラにより2～10バイトの機械語に変換される。

アセンブラ制御命令はアセンブラの処理を制御するステートメントで、機械語には変換されない。メモリ領域の確保、データの設定、値の割当て、領域の宣言などを行う。

コメント文はプログラム中に注釈を入れるためのものであり、第1カラムが\*で始まる。

## 11.2 アセンブラの書式

3種類のステートメントのうち、実行命令とアセンブラ制御命令を総称したものを命令ステートメントとよぶことにする。命令ステートメントは図11.2に示す4つのフィールド

	ラベル フィールド	演算 フィールド	オペランド フィールド	コメントフィールド
例	LABEL 1	ADD. B MOVE. L RTS	(A0), D1 D1, D2	COMMENT HERE COPY (DATA REGISTER) RETURN

■ 11.2 命令ステートメントの構成

から構成される。68000 アセンブラの記述形式はフリーフォーマットであり、各フィールド間は1つ以上の空白により区別される。次に各フィールドに関する機能と規則を説明する。

### 11.2.1 ラベルフィールド

ラベルはジャンプ先番地、データ領域番地などを指定するための記号名称であり、ラベルフィールドで定義され、オペランドによって参照される。ラベルは実行命令すべてにつけることができるが、アセンブラ制御命令にはラベルをつけることができない命令がある。ラベルは第1カラムから書くことになっている。また省略する場合には第1カラムを空白

にしなければならない。

### 11.2.2 演算フィールド

演算フィールドでは実行命令やアセンブラ制御命令のニモニックを記入する。実行命令は演算フィールドのニモニックとオペランドフィールドの記述に基づいて2~10バイトの機械語に変換される。データ定義命令 (DC: Define Constant) ではオペランドの値がメモリにセットされる。

実行命令のニモニックにはアドレス形式により変化形をもつものがある。変化形は命令のニモニックに A (アドレス), I (イミディエイト), Q (クイック), M (メモリ), X (エクステンド) をつけることによって作ることができる。このうち M と X の場合は必ずつけなければならないが, A, I, Q については基本形を用いても、アセンブラが自動的に適切な変化形を選択する。しかしプログラムのミスを少なくするためには変化形まで明記しておくことが望ましい<sup>†</sup>。また、たとえば

ADD #5, D1

のようにイミディエイトデータが小さい場合、指定がないとアセンブラは自動的に“ADDQ”を選択するので注意が必要である。実行命令の基本形と変化形の関係を表 11.1 に示す。

表 11.1 命令の変化形

命令の基本形	変 化 形	
	自動的に変化	明記が必要
ADD	ADDA, ADDI, ADDQ	ADDX
AND	ANDI	—
CMP	CMPA, CMPI	CMPM
EOR	EORI	—
MOVE	MOVEA, MOVEQ	—
NEG	—	NEGX
OR	ORI	—
SUB	SUBA, SUBI, SUBQ	SUBX
ROL, ROR	—	ROXL, ROXR

実行命令に対しては扱うデータサイズが、バイト (8 ビット)、ワード (16 ビット)、■  
ングワード (32 ビット) のいずれなのかを指定する必要がある。これらは命令のニモニ  
ックのあとに続けて .B (バイト)、.W (ワード)、.L (ロングワード) を書くことによ

<sup>†</sup> 本書のプログラム例はすべて変化形まで明記した。

って指定する。データサイズ指定は省略可能であり、その場合には原則としてワードが選ばれる。ただし、命令によっては暗黙的にバイト、ロングワードを選ぶものもあるので、各命令の項あるいは付録を参照されたい。

### 11.2.3 オペランド フィールド

オペランドフィールドには、命令で定められた個数のオペランドを、定められた順番で記入する。オペランドが2つある場合にはコンマ“,”で区切る。空白を途中に入れると以後の記述がコメントとみなされるので注意する必要がある。またオペランドが2つある場合には第1オペランドがソースオペランドを、第2オペランドがデスティネーションオペ

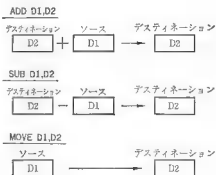


図 11.3 2つのオペランド演算の対応

ランドを表す。図11.3には68000の代表的な命令、ADD, SUB, MOVE命令について、オペランドの記述方法と演算のしかたを対応させて示した。

### 11.2.4 コメントフィールド

コメントフィールドはステートメントに注釈をつけたいときに使用するものであり省略できる。コメントフィールドには英数字のほかに空白を含め特殊文字が使用できる。

## 11.3 データ形式

アセンブラで使えるデータ形式には、数値定数、文字定数、記号、式の4種類がある。これらの記述の形式を次に述べる。

### 11.3.1 数値定数

数値定数は0～9の数字、およびA～Fの文字を組み合わせて得られる値である。以下に示すとおりの数値定数が記述できる。

- a. 10進定数 10進定数は0～9までの数字を組み合わせて構成される。数値の前に

“+” または “-” の符号をつけることにより正または負の値を表現できる。符号がない場合には正の値を表す。

135	10 進で 135 を表す
+348	10 進で 348 を表す
-79	10 進で -79 を表す

**b. 16 進定数** 16 進定数は 0~9 の数字および A~F の文字を組み合わせて作られ、先頭に “\$” をつけて区別する。なお日本国内では “\$” を用いる場合もある。16 進数に符号をつけることはできない。

\$10	10 進の 16 に等しい
\$A000	10 進の 40960 に等しい

**c. 2 進定数** 2 進定数は 0,1 の数字で構成され、先頭に “%” をつけて区別する。2 進数に符号をつけることはできない。

%10011101	10 進の 157 に等しい
%00100101	10 進の 37 に等しい

### 11.3.2 文 字 定 数

文字定数は英数字および特殊文字のほとんどが使用できる。文字定数の指定は ‘ (アポストロフィ) で囲むことにより行う。文字定数は次の例のように表す。

'AB8X'	文字列	AB8X を表す
'#%, SP'	文字列	#%, SP を表す
'AP'&+'	文字列	AP'&+ を表す

### 11.3.3 記 号

記号は、英字 [A]~[Z]、数字 [0]~[9]、記号 [@], [\$], [.] (ピリオド) からなる文字列であり、先頭は英字または記号 [@], [.] (ピリオド) のいずれかでなければならない。文字列の最大長はアセンブラの種類によって異なるが、8 字から 30 字程度である。記号のうち

A0, A1, ..., A7, D0, D1, ..., D7, CCR, SR, SP, USP

はアセンブラが予約語として使用するため、ラベルとして使うことはできない。なおアセンブラによっては別の予約語を使うものもある。記号の値は、ラベルフィールドに現れた

とき定義される。記号はラベルフィールドに2回以上現れてはならない。

記号に与えられた値は次のように決る。

**a. 実行命令および制御命令 (EQU を除く) のラベルとして現れたとき** このときのロケーションカウンタ (アセンブラが仮に決めるプログラムカウンタ) の値が記号に与えられる。この値は一般には相対値であるため、このような記号を相対値記号とよぶ<sup>†1</sup>。

**b. EQU 命令のラベルとして現れたとき** この命令のオペランド部の値が与えられる。記号の特性は、オペランドが数値定数あるいはそれを演算したものであれば絶対値記号 (値が確定) になり、相対値記号をオペランドとしていれば相対値記号となる。

これまで述べた以外に特別な記号として \* (アスタリスク) がある。これは \* が現れているステートメントの先頭のロケーションカウンタ<sup>†2</sup>の値を示すものであり、相対値記号として扱われるラベルフィールドに使うことはできない。

#### 11.3.4 式

オペランドフィールドには式を書くことができる。アセンブラは式の値を計算しその結果をオペランドとする。演算子としては + と - が許されているが、乗算/除算まで許すものもある。

式には数値定数および記号を使うことができる。記号には前に述べたように絶対値と相対値記号があり、同様に式にも絶対値式と相対値式の区別がある。絶対値式は数値定数および絶対値記号だけを含む式であり、相対値式は絶対値式と相対値記号を含む式である。また相対値記号の和は許されないが、差は絶対値式として扱われる。次に式の例を示す。ここで ABS 1, ABS 2 は EQU で定義された絶対値記号であり、REL 1 は実行命令のラベルにつけられた相対値記号であるとする。

[絶対値式の例]

1234

ABS 1 + ABS 2 - 100

[相対値式の例]

REL 1 - ABS 1 + 200

\* + ABS 1

<sup>†1</sup> これは 11.6 節で述べるリネージなプログラミングに関連しており、リネージのときに絶対アドレスが確定する。

<sup>†2</sup> ロケーションカウンタはアセンブラプログラムの中で定義された仮のプログラムカウンタである。

## 11.4 実効アドレスとアドレス形式

68000で利用できるアドレス形式は14種類あるが、これをアセンブラで記述するという立場からみると13種類に分類することができる。機械語レベルの分類との対応を表11.2に示す。アセンブラレベルの分類のうち1~12番目までがオペランドフィールドにアドレス形式を明記するものである。一方13番目のインプライド形式は、演算フィールドで自動的にオペランドが決るものあるいはオペランドのないものであって、オペランドフィールドには何も書かない。この形式をとる命令には、RTE, RTS, RTR, RESET, TRAPV, NOPがある。次にオペランドを明記するアドレス形式の記述のしかたを述べる。

表 11.2 アドレス形式の分類

No.	機械語レベルの分類	No.	アセンブラレベルの分類
1	データレジスタ直接形式	1	データレジスタ直接形式
2	アドレスレジスタ直接形式	2	アドレスレジスタ直接形式
3	アドレスレジスタ間接形式	3	アドレスレジスタ間接形式
4	ポストインクリメントアドレス レジスタ間接形式	4	ポストインクリメントアドレス レジスタ間接形式
5	プリデクリメントアドレス レジスタ間接形式	5	プリデクリメントアドレス レジスタ間接形式
6	ディスプレースメント付アドレス レジスタ間接形式	6	ディスプレースメント付アドレス レジスタ間接形式
7	インデックス付アドレス レジスタ間接形式	7	インデックス付アドレス レジスタ間接形式
8	長絶対アドレス形式	8	絶対アドレス形式
9	短絶対アドレス形式		
10	ディスプレースメント付プログラム カウンタ相対形式	9	ディスプレースメント付 プログラムカウンタ相対形式
11	インデックス付プログラム カウンタ相対形式	10	インデックス付プログラム カウンタ相対形式
12	イミディエイトデータ形式	11	イミディエイトデータ形式
13	クイックイミディエイトデータ形式		
14	インプライド形式	12	CCR/SR 形式
		13	インプライド形式



### 11.4.1 データレジスタ直接形式

オペランドフィールドで指定したデータレジスタが実効アドレスとなる。アセンブラ表現は  $D_n$  ( $n=0\sim7$ ) である。下線は当該アドレス形式を表す (以下同様)。

[例]    `ADD .W    A1, D2    [D2+A1→D2]`  
          `MOVE .B    D0, D5    [D0→D5]`

### 11.4.2 アドレスレジスタ直接形式

指定したアドレスレジスタが実効アドレスとなる。アセンブラ表現は  $A_n$  ( $n=0\sim7$ ) あるいは  $SP$  である。  $SP$  は  $A7$  または  $A7'$  と同じであり、システムスタックポインタを指す、ユーザシステムスタックポインタが使われるか、スーパーバイザシステムスタックポインタが使われるかは実行時のプロセッサのプログラム実行状態によって決る。また、この形式ではバイトサイズのデータを扱うことはできない。

なお特殊な命令として、スーパーバイザ状態の中でユーザスタックを操作するための `MOVE` 命令がある。この場合だけはオペランドに `USP` と書きユーザスタックポインタを指すことができる。

[例]    `SUBA .W    A1, A2    [A2-A1→A2]`  
          `MOVE .L    USP, A1    [USP→A1]`

### 11.4.3 アドレスレジスタ間接形式

アドレスレジスタの内容が指すメモリのデータが実効アドレスとなる。記述形式は  $(A_n)$  である ( $n=0\sim7$ )。

[例]    `MOVE .B    (A1), D2    [(A1)→D2]`  
          `TST .L    (A1)    [(A1)-0]`

### 11.4.4 ポストインクリメントアドレスレジスタ間接形式

アドレスレジスタの内容が指すメモリのデータが実行の対象となる。実行後アドレスレジスタには、扱ったデータサイズに従って 1 (バイト)、2 (ワード)、4 (ロングワード) が加算される。記述形式は  $(A_n)+$  である ( $n=0\sim7$ )。

[例]    `ADD .B    (A3)+, D5    [(D5)+A3→D5, A3+1→A3]`

## 11.4.5 プリデクリメントアドレスレジスタ間接形式

指定したアドレスレジスタから、あらかじめ扱うデータサイズに従って、1, 2, または 4 を引いたあと、アドレスレジスタ間接形式の処理をする。記述形式は  $-(An)$  である ( $n=0\sim7$ )。

【例】    `MOVE .W    -(A4), -(A5)`  
           `[A4-2→A4, A5-2→A5, (A4)→(A5)]`

## 11.4.6 ディスプレースメント付アドレスレジスタ間接形式

指定したアドレスレジスタと命令の中のディスプレースメントを加えた値が指すメモリの内容が実効アドレスとなる。ディスプレースメントは 16 ビットの符号付整数として扱われるため、 $-32768\sim+32767$  の範囲が指定できる。記述形式は  $d(An)$  である ( $n=0\sim7$ )。d は絶対値式でなければならない。

【例】    `DISP    EQU        4500`  
           `ADD .W        DISP+8 (A1), D2`  
           (この場合の d の値は 4508 となる)

## 11.4.7 インデックス付アドレスレジスタ間接形式

指定したアドレスレジスタとインデックスレジスタ、およびディスプレースメントを加えた値が指すメモリの内容が実行の対象となる。ディスプレースメント d は 8 ビットの符号付整数であり、 $-128\sim127$  の範囲が指定できる。記述形式は  $d(An, Ri)$  または  $d(An, Ri.W)$  または  $d(An, Ri.L)$  である ( $n=0\sim7$ )。Ri はインデックスレジスタであり、アドレスレジスタ Ai およびデータレジスタ Di の中から任意の 1 個をこれに当てることができる ( $i=0\sim7$ )。W または .L を指定することにより、インデックスレジスタの下位 16 ビットを使うか、32 ビットすべてを使うかを選ぶことができる。何も指定しないときは .W が選ばれる。W を指定したときは 16 ビット目(最上位ビット)を符号拡張したものが使われる。d は絶対値式でなければならない。

【例】    `DISP    EQU        14`  
           `AND .B        D0, 60 (A1, A2 .L)`  
           `OR .L        DISP-70 (A0, D1 .W), D5`  
           (この場合の d の値は -56 となる)

## 11.4.8 絶対アドレス形式

オペランドフィールドの値が指すメモリの内容が実行の対象となる。記述形式は  $m$  である。ここで、 $m$  は絶対値式または相対値式である。この命令はアドレス空間の全領域を指定できる。もし  $0 \sim 32767$  までのアドレスが指定された場合には短絶対アドレス形式となるため、命令の長さを短くすることができる。

【例】 ADR 1: 絶対値が EQU でセットされている。

```
ADD .W    2154, D1
ADD .W    ADR 1, D1
ADD .W    *+ADR 1, D1
```

## 11.4.9 ディスプレースメント付プログラムカウンタ相対形式

オペランドフィールドで指定したラベル、あるいは式の値が指すメモリの内容が実効アドレスとなる。記述形式は  $m$  であり、 $m$  には相対値式が入る。命令の中のディスプレースメントは 16 ビットの符号付整数であるから、この形式で指すことのできるメモリ範囲は、命令の先頭アドレス (=  $*$ ) に対して  $-32766 \sim 32769$  である。これはプログラムカウンタが  $*+2$  を指しているためである。

【例】 ADR 1: EQU で与えられた絶対値

```
LBL 1: 実行命令のラベル
DBNE    *+ADR 1
BRA     LBL 1
```

## 11.4.10 インデックス付プログラムカウンタ相対形式

プログラムカウンタと、インデックスレジスタと、ディスプレースメントを加えた値が指すメモリの内容が実効アドレスとなる。記述形式は  $d(Ri)$  または  $d(Ri, W)$  または  $d(Ri, L)$  である。インデックスレジスタ  $Ri$  の指定のしかたはインデックス付アドレスレジスタ間接形式の場合と同じである。 $d$  には相対値式を書くが、式の値は命令の先頭アドレスに対して  $-126 \sim +129$  の範囲に入っていないとてはならない。

【例】 LBL: ラベル

```
SUB .L    LBL+10(A0), D2 .W
```

## 11.4.11 イミディエイトデータ形式

オペランドがそのまま実行の対象となる。記述形式は  $\#m$  である。  $m$  は絶対値式、文字定数、あるいはアドレスの指定しているプログラムの中の相対値式である。指定できる値の大きさは、オペランドフィールドで指定するデータサイズにより決る。

【例】 ADR 1: EQU で定義された絶対値

```

MOVE .W  #100, D1      10進数
MOVE .L  #'AXZ 3', D1   文字定数
MOVE .W  #ADR 1, D1     絶対値式

```

## 11.4.12 CCR/SR 形式

ステータスレジスタ (SR) またはコンディションコードレジスタ (CCR:SR の下8ビット) が実効アドレスとなる。記述形式は SR あるいは CCR である。この形式は ANDI, ORI, EORI の第2オペランド、および MOVE で使うことができる。

```

【例】  ANDI .B  #$1D, CCR
        MOVE    SR, D0

```

## 11.5 アセンブラ制御命令

ここではいろいろなアセンブラ制御命令のうちで基本的なものについて説明する。

## (1) ORG &lt;絶対アドレス&gt; (Absolute Section Origin)

オペランドフィールドに書かれた値を ORG に続くステートメントの絶対アドレスとする。この命令はプログラムの途中に複数回現れてもよい。この命令にラベルをつけることはできない。

【例】 ORG \$1000

## (2) &lt;記号&gt; EQU &lt;式&gt; (Equate Symbol Value)

ラベルフィールドの記号に式の値を与える。式は絶対値式がふつうであるが、すでに現れた記号あるいはロケーションカウンタ (\*) を用いた相対値式も記述できる。ラベルを省略することはできない。

```

【例】  EAST    EQU    3
        WEST    EQU    EAST
        HERE    EQU    *

```

## (3) DC &lt;式&gt;, &lt;式&gt;, ... (Define Constant)

式を計算した値をメモリに定数データとして格納する命令であり、カンマによって複数のデータ定義が可能である。DCに続いてデータサイズを .B, .W, .L で定義でき、省略時には .W が仮定される。この命令にはラベルをつけてもよい。

```
[例] VAL1    DC .W 31, $A, 'XYZ'
          DC .B  'ABC'
```

## (4) DS &lt;絶対値式&gt; (Define Storage)

絶対値式で与えられた大きさのエリアを、データサイズを単位としてメモリに確保する。データサイズは .B, .W, .L で指定でき、省略時には .W が仮定される。命令にラベルをつけることが可能である。

```
[例] AREA   DS .L 8
```

これまでに述べたアセンブラ制御命令および実行命令を用いたプログラム例を図 11.4 に示す。この例は先頭が TBL1 で示されるテーブル 1 を、先頭が TBL2 で示されるテーブル 2 に加算するものである。またテーブル 1 のデータが 0 になったらそこで加算を終了する。MCOUNT はテーブル 1 の値が 0 にならない場合に終了すべき回数を設定している。ここではアセンブラ制御命令として EQU, ORG, DS, DC を用いており、このほかにプログラムの開始を示す NAM, 終了を示す END を使っている。

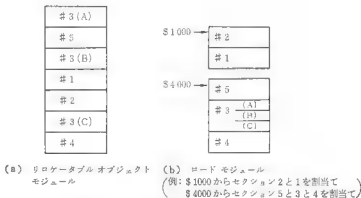
```

00001          NAM      SAMPLE1
00002          *-----*
00003          * SAMPLE PROGRAM (ADD TABLE 1) *
00004          * TABLE1 + TABLE2 -> TABLE2 *
00005          *-----*
00006          MCOUNT    EQU 20          MAX TABLE LENGTH
00007          TBL2      EQU $2000      TABLE2 ADDRESS
00008          MONITOR   EQU 10          MONITOR TRAP NO.
00009          *
00010          ORG      $1000          PROGRAM ADDRESS
00011          *
00100 0000 2270 0000 102A 00012          MOVE.L #TBL1,A1      LOAD TABLE1 ADDRESS INTO A1
00101 0000 247C 0000 2000 00013          MOVE.L #TBL2,A2      LOAD TABLE2 ADDRESS INTO A2
00102 0000 323C 0014 00014          MOVE.W #MCOUNT,D1     LOAD MAX COUNT INTO D1
00103 0000 4242 0000 00015          CLW.D2              INITIALIZE D2(COUNTER)
00104 0000 3619 0000 00016          START MOVE.W (A1)+,D3     LOAD TABLE1 INTO D3
00105 0000 6700 000A 00017          BRQ FINISH          IF DATA = 0 THEN END
00106 0000 5242 0000 00018          ADDQ.W #1,D2          COUNTER INCREMENT
00107 0000 D75A 0000 00019          ADD.W D3,(A2)+      ADD AND STORE TABLE2
00108 0000 51C9 FFF4 00020          DBR4 D1,START      MAX COUNT CHECK
00109          *
00110 0000 33C2 0000 102B 00021          FINISH MOVE.W D2,COUNT     STORE COUNT VALUE
00111 0000 4E44 0000 00022          TRAP MONITOR      GO TO MONITOR ROUTINE
00112          *
00113 0000 0000 0002 00023          COUNT DS.W 1
00114          *
00115 0000 0087 0000 01CB 00024          TBL1 DC.W 135,36,456,8,2,127,87,54,0
00116 0000 0057 003B 0000 00025          *
00117          *
00118          *
00119          *
00120          *
00121          *
00122          *
00123          *
00124          *
00125          *
00126          *
00127          *
00128          *
00129          *
00130          *
00131          *
00132          *
00133          *
00134          *
00135          *
00136          *
00137          *
00138          *
00139          *
00140          *
00141          *
00142          *
00143          *
00144          *
00145          *
00146          *
00147          *
00148          *
00149          *
00150          *
00151          *
00152          *
00153          *
00154          *
00155          *
00156          *
00157          *
00158          *
00159          *
00160          *
00161          *
00162          *
00163          *
00164          *
00165          *
00166          *
00167          *
00168          *
00169          *
00170          *
00171          *
00172          *
00173          *
00174          *
00175          *
00176          *
00177          *
00178          *
00179          *
00180          *
00181          *
00182          *
00183          *
00184          *
00185          *
00186          *
00187          *
00188          *
00189          *
00190          *
00191          *
00192          *
00193          *
00194          *
00195          *
00196          *
00197          *
00198          *
00199          *
00200          *
00201          *
00202          *
00203          *
00204          *
00205          *
00206          *
00207          *
00208          *
00209          *
00210          *
00211          *
00212          *
00213          *
00214          *
00215          *
00216          *
00217          *
00218          *
00219          *
00220          *
00221          *
00222          *
00223          *
00224          *
00225          *
00226          *
00227          *
00228          *
00229          *
00230          *
00231          *
00232          *
00233          *
00234          *
00235          *
00236          *
00237          *
00238          *
00239          *
00240          *
00241          *
00242          *
00243          *
00244          *
00245          *
00246          *
00247          *
00248          *
00249          *
00250          *
00251          *
00252          *
00253          *
00254          *
00255          *
00256          *
00257          *
00258          *
00259          *
00260          *
00261          *
00262          *
00263          *
00264          *
00265          *
00266          *
00267          *
00268          *
00269          *
00270          *
00271          *
00272          *
00273          *
00274          *
00275          *
00276          *
00277          *
00278          *
00279          *
00280          *
00281          *
00282          *
00283          *
00284          *
00285          *
00286          *
00287          *
00288          *
00289          *
00290          *
00291          *
00292          *
00293          *
00294          *
00295          *
00296          *
00297          *
00298          *
00299          *
00300          *
00301          *
00302          *
00303          *
00304          *
00305          *
00306          *
00307          *
00308          *
00309          *
00310          *
00311          *
00312          *
00313          *
00314          *
00315          *
00316          *
00317          *
00318          *
00319          *
00320          *
00321          *
00322          *
00323          *
00324          *
00325          *
00326          *
00327          *
00328          *
00329          *
00330          *
00331          *
00332          *
00333          *
00334          *
00335          *
00336          *
00337          *
00338          *
00339          *
00340          *
00341          *
00342          *
00343          *
00344          *
00345          *
00346          *
00347          *
00348          *
00349          *
00350          *
00351          *
00352          *
00353          *
00354          *
00355          *
00356          *
00357          *
00358          *
00359          *
00360          *
00361          *
00362          *
00363          *
00364          *
00365          *
00366          *
00367          *
00368          *
00369          *
00370          *
00371          *
00372          *
00373          *
00374          *
00375          *
00376          *
00377          *
00378          *
00379          *
00380          *
00381          *
00382          *
00383          *
00384          *
00385          *
00386          *
00387          *
00388          *
00389          *
00390          *
00391          *
00392          *
00393          *
00394          *
00395          *
00396          *
00397          *
00398          *
00399          *
00400          *
00401          *
00402          *
00403          *
00404          *
00405          *
00406          *
00407          *
00408          *
00409          *
00410          *
00411          *
00412          *
00413          *
00414          *
00415          *
00416          *
00417          *
00418          *
00419          *
00420          *
00421          *
00422          *
00423          *
00424          *
00425          *
00426          *
00427          *
00428          *
00429          *
00430          *
00431          *
00432          *
00433          *
00434          *
00435          *
00436          *
00437          *
00438          *
00439          *
00440          *
00441          *
00442          *
00443          *
00444          *
00445          *
00446          *
00447          *
00448          *
00449          *
00450          *
00451          *
00452          *
00453          *
00454          *
00455          *
00456          *
00457          *
00458          *
00459          *
00460          *
00461          *
00462          *
00463          *
00464          *
00465          *
00466          *
00467          *
00468          *
00469          *
00470          *
00471          *
00472          *
00473          *
00474          *
00475          *
00476          *
00477          *
00478          *
00479          *
00480          *
00481          *
00482          *
00483          *
00484          *
00485          *
00486          *
00487          *
00488          *
00489          *
00490          *
00491          *
00492          *
00493          *
00494          *
00495          *
00496          *
00497          *
00498          *
00499          *
00500          *
00501          *
00502          *
00503          *
00504          *
00505          *
00506          *
00507          *
00508          *
00509          *
00510          *
00511          *
00512          *
00513          *
00514          *
00515          *
00516          *
00517          *
00518          *
00519          *
00520          *
00521          *
00522          *
00523          *
00524          *
00525          *
00526          *
00527          *
00528          *
00529          *
00530          *
00531          *
00532          *
00533          *
00534          *
00535          *
00536          *
00537          *
00538          *
00539          *
00540          *
00541          *
00542          *
00543          *
00544          *
00545          *
00546          *
00547          *
00548          *
00549          *
00550          *
00551          *
00552          *
00553          *
00554          *
00555          *
00556          *
00557          *
00558          *
00559          *
00560          *
00561          *
00562          *
00563          *
00564          *
00565          *
00566          *
00567          *
00568          *
00569          *
00570          *
00571          *
00572          *
00573          *
00574          *
00575          *
00576          *
00577          *
00578          *
00579          *
00580          *
00581          *
00582          *
00583          *
00584          *
00585          *
00586          *
00587          *
00588          *
00589          *
00590          *
00591          *
00592          *
00593          *
00594          *
00595          *
00596          *
00597          *
00598          *
00599          *
00600          *
00601          *
00602          *
00603          *
00604          *
00605          *
00606          *
00607          *
00608          *
00609          *
00610          *
00611          *
00612          *
00613          *
00614          *
00615          *
00616          *
00617          *
00618          *
00619          *
00620          *
00621          *
00622          *
00623          *
00624          *
00625          *
00626          *
00627          *
00628          *
00629          *
00630          *
00631          *
00632          *
00633          *
00634          *
00635          *
00636          *
00637          *
00638          *
00639          *
00640          *
00641          *
00642          *
00643          *
00644          *
00645          *
00646          *
00647          *
00648          *
00649          *
00650          *
00651          *
00652          *
00653          *
00654          *
00655          *
00656          *
00657          *
00658          *
00659          *
00660          *
00661          *
00662          *
00663          *
00664          *
00665          *
00666          *
00667          *
00668          *
00669          *
00670          *
00671          *
00672          *
00673          *
00674          *
00675          *
00676          *
00677          *
00678          *
00679          *
00680          *
00681          *
00682          *
00683          *
00684          *
00685          *
00686          *
00687          *
00688          *
00689          *
00690          *
00691          *
00692          *
00693          *
00694          *
00695          *
00696          *
00697          *
00698          *
00699          *
00700          *
00701          *
00702          *
00703          *
00704          *
00705          *
00706          *
00707          *
00708          *
00709          *
00710          *
00711          *
00712          *
00713          *
00714          *
00715          *
00716          *
00717          *
00718          *
00719          *
00720          *
00721          *
00722          *
00723          *
00724          *
00725          *
00726          *
00727          *
00728          *
00729          *
00730          *
00731          *
00732          *
00733          *
00734          *
00735          *
00736          *
00737          *
00738          *
00739          *
00740          *
00741          *
00742          *
00743          *
00744          *
00745          *
00746          *
00747          *
00748          *
00749          *
00750          *
00751          *
00752          *
00753          *
00754          *
00755          *
00756          *
00757          *
00758          *
00759          *
00760          *
00761          *
00762          *
00763          *
00764          *
00765          *
00766          *
00767          *
00768          *
00769          *
00770          *
00771          *
00772          *
00773          *
00774          *
00775          *
00776          *
00777          *
00778          *
00779          *
00780          *
00781          *
00782          *
00783          *
00784          *
00785          *
00786          *
00787          *
00788          *
00789          *
00790          *
00791          *
00792          *
00793          *
00794          *
00795          *
00796          *
00797          *
00798          *
00799          *
00800          *
00801          *
00802          *
00803          *
00804          *
00805          *
00806          *
00807          *
00808          *
00809          *
00810          *
00811          *
00812          *
00813          *
00814          *
00815          *
00816          *
00817          *
00818          *
00819          *
00820          *
00821          *
00822          *
00823          *
00824          *
00825          *
00826          *
00827          *
00828          *
00829          *
00830          *
00831          *
00832          *
00833          *
00834          *
00835          *
00836          *
00837          *
00838          *
00839          *
00840          *
00841          *
00842          *
00843          *
00844          *
00845          *
00846          *
00847          *
00848          *
00849          *
00850          *
00851          *
00852          *
00853          *
00854          *
00855          *
00856          *
00857          *
00858          *
00859          *
00860          *
00861          *
00862          *
00863          *
00864          *
00865          *
00866          *
00867          *
00868          *
00869          *
00870          *
00871          *
00872          *
00873          *
00874          *
00875          *
00876          *
00877          *
00878          *
00879          *
00880          *
00881          *
00882          *
00883          *
00884          *
00885          *
00886          *
00887          *
00888          *
00889          *
00890          *
00891          *
00892          *
00893          *
00894          *
00895          *
00896          *
00897          *
00898          *
00899          *
00900          *
00901          *
00902          *
00903          *
00904          *
00905          *
00906          *
00907          *
00908          *
00909          *
00910          *
00911          *
00912          *
00913          *
00914          *
00915          *
00916          *
00917          *
00918          *
00919          *
00920          *
00921          *
00922          *
00923          *
00924          *
00925          *
00926          *
00927          *
00928          *
00929          *
00930          *
00931          *
00932          *
00933          *
00934          *
00935          *
00936          *
00937          *
00938          *
00939          *
00940          *
00941          *
00942          *
00943          *
00944          *
00945          *
00946          *
00947          *
00948          *
00949          *
00950          *
00951          *
00952          *
00953          *
00954          *
00955          *
00956          *
00957          *
00958          *
00959          *
00960          *
00961          *
00962          *
00963          *
00964          *
00965          *
00966          *
00967          *
00968          *
00969          *
00970          *
00971          *
00972          *
00973          *
00974          *
00975          *
00976          *
00977          *
00978          *
00979          *
00980          *
00981          *
00982          *
00983          *
00984          *
00985          *
00986          *
00987          *
00988          *
00989          *
00990          *
00991          *
00992          *
00993          *
00994          *
00995          *
00996          *
00997          *
00998          *
00999          *
01000          *
01001          *
01002          *
01003          *
01004          *
01005          *
01006          *
01007          *
01008          *
01009          *
01010          *
01011          *
01012          *
01013          *
01014          *
01015          *
01016          *
01017          *
01018          *
01019          *
01020          *
01021          *
01022          *
01023          *
01024          *
01025          *
01026          *
01027          *
01028          *
01029          *
01030          *
01031          *
01032          *
01033          *
01034          *
01035          *
01036          *
01037          *
01038          *
01039          *
01040          *
01041          *
01042          *
01043          *
01044          *
01045          *
01046          *
01047          *
01048          *
01049          *
01050          *
01051          *
01052          *
01053          *
01054          *
01055          *
01056          *
01057          *
01058          *
01059          *
01060          *
01061          *
01062          *
01063          *
01064          *
01065          *
01066          *
01067          *
01068          *
01069          *
01070          *
01071          *
01072          *
01073          *
01074          *
01075          *
01076          *
01077          *
01078          *
01079          *
01080          *
01081          *
01082          *
01083          *
01084          *
01085          *
01086          *
01087          *
01088          *
01089          *
01090          *
01091          *
01092          *
01093          *
01094          *
01095          *
01096          *
01097          *
01098          *
01099          *
01100          *
01101          *
01102          *
01103          *
01104          *
01105          *
01106          *
01107          *
01108          *
01109          *
01110          *
01111          *
01112          *
01113          *
01114          *
01115          *
01116          *
01117          *
01118          *
01119          *
01120          *
01121          *
01122          *
01123          *
01124          *
01125          *
01126          *
01127          *
01128          *
01129          *
01130          *
01131          *
01132          *
01133          *
01134          *
01135          *
01136          *
01137          *
01138          *
01139          *
01140          *
01141          *
01142          *
01143          *
01144          *
01145          *
01146          *
01147          *
01148          *
01149          *
01150          *
01151          *
01152          *
01153          *
01154          *
01155          *
01156          *
01157          *
01158          *
01159          *
01160          *
01161          *
01162          *
01163          *
01164          *
01165          *
01166          *
01167          *
01168          *
01169          *
01170          *
01171          *
01172          *
01173          *
01174          *
01175          *
01176          *
01177          *
01178          *
01179          *
01180          *
01181          *
01182          *
01183          *
01184          *
01185          *
01186          *
01187          *
01188          *
01189          *
01190          *
01191          *
01192          *
01193          *
01194          *
01195          *
01196          *
01197          *
01198          *
01199          *
01200          *
01201          *
01202          *
01203          *
01204          *
01205          *
01206          *
01207          *
01208          *
01209          *
01210          *
01211          *
01212          *
01213          *
01214          *
01215          *
01216          *
01217          *
01218          *
01219          *
01220          *
01221          *
01222          *
01223          *
01224          *
01225          *
01226          *
01227          *
01228          *
01229          *
01230          *
01231          *
01232          *
01233          *
01234          *
01235          *
01236          *
01237          *
01238          *
01239          *
01240          *
01241          *
01242          *
01243          *
01244          *
01245          *
01246          *
01247          *
01248          *
01249          *
01250          *
01251          *
01252          *
01253          *
01254          *
01255          *
01256          *
01257          *
01258          *
01259          *
01260          *
01261          *
01262          *
01263          *
01264          *
01265          *
01266          *
01267          *
01268          *
01269          *
01270          *
01271          *
01272          *
01273          *
01274          *
01275          *
01276          *
01277          *
01278          *
01279          *
01280          *
01281          *
01282          *
01283          *
01284          *
01285          *
01286          *
01287          *
01288          *
01289          *
01290          *
01291          *
01292          *
01293          *
01294          *
01295          *
01296          *
01297          *
01298          *
01299          *
01300          *
01301          *
01302          *
01303          *
01304          *
01305          *
01306          *
01307          *
01308          *
01309          *
01310          *
01311          *
01312          *
01313          *
01314          *
01315          *
01316          *
01317          *
01318          *
01319          *
01320          *
01321          *
01322          *
01323          *
01324          *
01325          *
01326          *
01327          *
01328          *
01329          *
01330          *
01331          *
01332          *
01333          *
01334          *
01335          *
01336          *
01337          *
01338          *
01339          *
01340          *
01341          *
01342          *
01343          *
01344          *
01345          *
01346          *
01347          *
01348          *
01349          *
01350          *
01351          *
01352          *
01353          *
01354          *
01355          *
01356          *
01357          *
01358          *
01359          *
01360          *
01361          *
01362          *
01363          *
01364          *
01365          *
01366          *
01367          *
01368          *
01369          *
01370          *
01371          *
01372          *
01373          *
01374          *
01375          *
01376          *
01377          *
01378          *
01379          *
01380          *
01381          *
01382          *
01383          *
01384          *
01385          *
01386          *
01387          *
01388          *
01389          *
01390          *
01391          *
01392          *
01393          *
01394          *
01395          *
01396          *
01397          *
01398          *
01399          *
01400          *
01401          *
01402          *
01403          *
01404          *
01405          *
01406          *
01407          *
01408          *
01409          *
01410          *
01411          *
01412          *
01413          *
01414          *
01415          *
01416          *
01417          *
01418          *
01419          *
01420          *
01421          *
01422          *
01423          *
01424          *
01425          *
01426          *
01427          *
01428          *
01429          *
01430          *
01431          *
01432          *
01433          *
01434          *
01435          *
01436          *
01437          *
01438          *
01439          *
01440          *
01441          *
01442          *
01443          *
01444          *
01445          *
01446          *
01447          *
01448          *
01449          *
01450          *
01451          *
01452          *
01453          *
01454          *
01455          *
01456          *
01457          *
01458          *
01459          *
01460          *
01461          *
01462          *
01463          *
01464          *
01465          *
01466          *
01467          *
01468          *
01469          *
01470          *
01471          *
01472          *
01473          *
01474          *
01475          *
01476          *
01477          *
01478          *
01479          *
01480          *
01481          *
01482          *
0
```

## 11.6 リロケートブルなプログラミング

これまで述べてきたプログラムは、ソースプログラムを記述するときにすでにアドレスが確定しているものと考えていた。しかし大きなシステムでは、プログラムをいくつかの単位に分割して作っておき、これらを適当に組み合わせて所望のプログラムを作る場合が多い。この利点は、一般に機能単位でプログラムを分割するので、複数のプログラマが同時に作業できること、デバッグや修正が容易であること、別のプログラムへの変更も単位プログラムの組合せを変えるだけで対応できること、などである。この場合ソースプログラムでは絶対アドレスを与えず、どこにでも配置できるようにしておくことが望ましい。このようにすることをリロケートブルなプログラミングとよぶ。

リロケートブルなプログラムの分割の単位はセクションである。セクションには番号がつけられており、各セクションはアセンブラにより仮のアドレスがついたオブジェクトモジュールになる。そしてリンカージェディタで、先頭のアドレスと、セクションの番号を結合したい順に指定すると、絶対アドレスの確定したロードモジュールが作成される。この様子を図 11.5 に示す。(a) はソースプログラムおよびこれをアセンブルしたオブジェクトモジュールにおけるセクションの並びを示している。セクション 3 のように 1 つのセクションがさらに分割されていてもよい。リンカージェディタで、セクション 1 と 2 を 2, 1 の順に \$1000 番地から割り当て、セクション 3, 4, 5, は 5, 3, 4 の順に \$4000 番地から割り当てるように指定する。するとロードモジュールは図の (b) のように、アドレス付



■ 11.5 メモリの割当て例。注) #n はセクション番号

けされて作成される。

リロケータブルなプログラムにするには新たなアセンブラ制御命令が必要である<sup>†</sup>。リロケータブルなセクションであることを宣言するには、**ORG** 命令の代りに **SCT** あるいは **SECTION** 命令を用いる。またリロケータブルなセクションでは、ベースレジスタを登録しておきベースレジスタ間接の方法で、アドレス指定をする。この場合まず **USING** 命令でどのアドレスレジスタをベースレジスタとするかをアセンブラに対し指定する。するとラベルを参照するアドレス形式は自動的にディスプレイメント付アドレスレジスタ間接形式、あるいはインデックス付アドレスレジスタ間接形式に変換される。なお **USING** 命令は、アセンブラ制御命令でありレジスタに実際の値を入れる機能はない。ベースレジスタに値を設定する命令はユーザが実行命令で記述しておかねばならない。リロケータブルにするための命令はこのほかにもいくつかあるが、ここではふれない。

---

<sup>†</sup> 小規模なアセンブラではサポートしていないものもある。

## 12 データ転送命令

データ転送命令は、メモリや68000の内部レジスタ間でデータを転送するのに用いられる。68000では、特に多くの種類のデータ転送命令を有していることが命令セットの特徴ともなっている。さらにバイト、ワード、ロングワードなどのオペランドサイズと14種のアドレス形式を組み合わせることにより、高速データ転送、大量データ転送など効率の良いデータ転送を実現できる。5章では各転送命令がとり得るオペランドサイズとオペレーションの概要を示した。この章では、転送命令のうち、LINK、UNLK命令以外の命令とその主要な応用例について説明する。LINK、UNLK命令は15章で述べる。

### 12.1 データ転送命令

a. MOVE (Move Data) 命令† MOVE命令はデータ転送命令のなかで最も基本的な命令である。この命令は、バイト(B)、ワード(W)、ロングワード(L)のデータを図12.1のようにメモリ、レジスタ間で転送する。このようにメモリ、レジスタの任意の組合せで転送ができること、付録の命令一覧表で示すように、ソースおよびデスティネーションのアドレス形式を独立に、かつ任意に指定できることが大きな特徴となっている。

MOVE命令ではデータ転送時にデータの正、負、0を判定して、その結果をコンディションコードレジスタのN、Zの各フラグに反映させる。一方、V、Cの各フラグは、リセットされる。X(Extend)フラグは影響を受けない。

データレジスタからメモリへのデータ転送のための命令の使用例を以下に示す。

---

† MOVE <ea1>, <ea2>    <ea1> ← <ea2>



		デスティネーション				
		メモリー	データレジスタ	アドレスレジスタ	コンディションコードレジスタ	ステータスレジスタ
ソース	メモリー	B W L	B W L	W L (MOVEA)	W (MOVE to CCR)	W (MOVE to SR)
	データレジスタ	B W L	B W L	W L (MOVEA)	W (MOVE to CCR)	W (MOVE to SR)
	アドレスレジスタ	B W L	B W L	W* L (MOVEA)		
	イミディエイトデータ	B W L	B W L	W L (MOVEA)	W (MOVE to CCR)	W (MOVE to SR)
	コンディションコードレジスタ					
	ステータスレジスタ	W (MOVE from SR)	W (MOVE from SR)			

B: バイト, W: ワード, L: ロングワード

図 12.1 データ転送命令のソースとデスティネーション

MOVE .B D0, D1      D0 の内容の最下位バイトを D1 へ転送する  
 MOVE .L D2, ADDR    D2 の内容を ADDR 番地へ転送する  
 MOVE .W D3, (A1)+    D3 の内容を A1 で指定されたメモリの番地へ転送し、その後 A1 を +2 する

ここで ADDR と記述されたラベルには、あらかじめメモリ番地を与えておき、アドレスレジスタ A1 にも、あらかじめメモリ番地がセットされているとする。

**b. MOVEA (Move Address) 命令\*** この命令はアドレスデータの転送に用いられるため、転送のデスティネーションはつねにアドレスレジスタとなる。転送データのサイズは、ワード、ロングワードのみである。各フラグは影響を受けない。

MOVEA 命令の機械語は、MOVE 命令においてデスティネーションオペランドをアドレスレジスタに指定したときのものと同じである。このため MOVE 命令でデスティネー

\* MOVEA <ea>, An    <ea> → An

ションオペランドにアドレスレジスタを指定すると、アセンブラにおいて MOVEA 命令に自動的に置き換る。なおデータサイズがワードの場合はデータの符号拡張が行われる。

本命令の例として、サブルーチンジャンプ時のレジスタの退避処理を次に示す。

JSR	SUBR	サブルーチン SUBR へ分岐する
SUBR	MOVEA, L A0, SAVE	A0 の内容を SAVE 番地へ退避する
	MOVEA, L SAVE, A0	SAVE 番地より A0 の値を回復する
	RTS	サブルーチンからの復帰

MOVE 命令と MOVEA 命令におけるデータ転送の様子をまとめて 図 12.2 に示す。(1) はアドレスレジスタで指定されたメモリ上の 32 ビットのデータをデータレジスタに転送する場合である。ADDR i 番地のデータはデータレジスタの上位ワードへ、ADDR i+2 番地のデータは下位ワードに移される。(2) はメモリ間のデータ転送の場合でアドレスレジスタを 2 個使って任意のメモリ間のデータ転送が行える。(3) はアドレスレジスタ

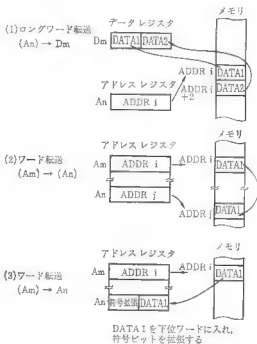
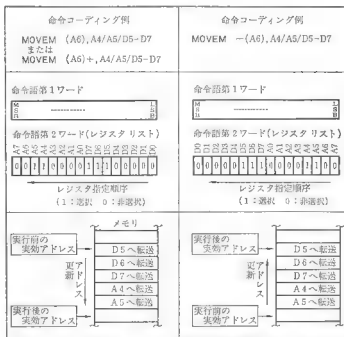


図 12.2 データ転送命令の例

へのデータ転送の場合 (MOVEA 命令) で、32 ビット データに符号が拡張されていることに注意されたい。

c. **MOVEM (Move Multiple Registers) 命令<sup>†1</sup>** MOVEM 命令は、図 12.3 に示されるように、命令語の第 2 ワードで指定されるレジスタ リストと第 1 ワードの実効アドレスを用いて、レジスタとメモリ間のデータ転送を行う。図 12.3 のレジスタ リスト上の各ビットの “1” 指定により選択されたデータ レジスタおよびアドレス レジスタと、実効アドレスを先頭アドレスとしたメモリとの間でデータ転送が行われる。レジスタ リスト上の各ビットで選択できるレジスタは、図 12.3 で示すようにメモリのアドレス更新方向によって異なるので注意が必要である。これについては後述する。この命令では、オペランドサイズをワード、およびロング ワードとすることができる。特にオペランドサイズとしてワ



■ 12.3 MOVEM 命令とレジスタ リスト

- †1 MOVEM <レジスタ リスト>, <ea> レジスタ リストで指定されたレジスタ → <ea>  
MOVEM <ea>, <レジスタ リスト> <ea> → レジスタ リストで指定されたレジスタ

†2 (次ページ) 一般の MOVE 命令では、オペランド長がワードのときにはデータレジスタは符号拡張されないのに注意されたい。

ードを指定しデスティネーションにデータレジスタを指定した場合、データレジスタには32ビットに符号拡張されたデータが転送される(10ページ)。

MOVEM 命令は、アドレス形式により、その動作形態が若干異なるので、それらについて以下に説明する。

(i) アドレスレジスタ間接、ディスプレースメント付/インテックス付アドレスレジスタ間接および長/短絶対アドレス形式 図12.3に示すように、レジスタデータの転送はD0, D1, ..., D7, A0, A1, ..., A7の順序で、メモリデータは命令で指定された先頭実効アドレスからアドレスが増加する順序で。データ転送が行われる。転送方向は、「レジスタからメモリへ」、および「メモリからレジスタへ」の両方向が可能である。

アドレスレジスタ間接、ディスプレースメント付アドレスレジスタ間接の各アドレス形式を用いてメモリからレジスタへの転送を行う場合、および実効アドレスを指定しているアドレスレジスタ自身にメモリからデータを転送した場合について説明する。68000ではMOVEM命令実行開始時に、上記アドレスレジスタの内容を内部の一時記憶用の特別なレジスタに移しておき、この一時記憶レジスタを用いて実効アドレス計算を行っている。したがって実効アドレスを指定しているアドレスレジスタへのデータ転送も矛盾なく行われる。

(ii) ディスプレースメント付/インテックス付プログラムカウンタ相対アドレス形式 レジスタおよびメモリのデータ転送の順序は(i)と同じである。しかしデータ転送の方向は、「メモリからレジスタへ」の転送のみ可能である。実効アドレスを記憶しているアドレスレジスタ(本命令ではデータレジスタも含まれる)に、メモリよりのデータを書き込む場合は、(i)と同じ理由で正常にデータ転送が行われる。

(iii) プリデクリメントアドレスレジスタ間接アドレス形式 このアドレス形式では、前記の場合とは逆に、レジスタデータの転送はA7, A6, ..., A0, D7, D6, ..., D0の順序である。メモリデータの転送は指定された実効アドレスからワードの場合は2、ロングワードの場合は4を減じたアドレスを先頭としてアドレスが減少する順序で行われる。転送方向は、「レジスタからメモリへ」のみが可能である。命令実行完了時の実効アドレス用のアドレスレジスタには、最後にデータを書き込んだメモリ番地が残っている。

(iv) ポストインクリメントアドレスレジスタ間接アドレス形式 データ転送が行われていくにつれて、実効アドレスが増加していくのは(i)と同様であるが、データ転送の方向は、「メモリからレジスタへ」のみが可能である。

このアドレス形式を用いて、実効アドレスを記憶しているアドレスレジスタへメモリか

らデータを転送すると、命令実行終了時には最後に読み出したメモリアドレスに2を加えたアドレス（ロングワードのときは4を加えたアドレス）が書き込まれる。

MOVEM 命令の使用例を以下に示す。

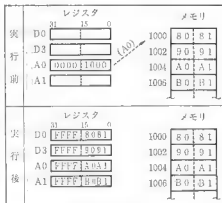
【例1】 メモリの \$1000 番地から、

8 バイトのデータを、図 12.4 のように D0, D3, A0, A1 の各レジスタへ 1 ワードずつ転送する。ただし A0 は実効アドレスを定めるアドレスレジスタとして用いる。MOVEM 命令実行後に各レジスタに転送されたデータは符号拡張されたものとなっている。

```

MOVEA .L    # $1000, A0
MOVEM .W    (A0), D0/D3/A0-A1
ORG         $1000
DC .L       $80819091
DC .L       $A0A1B0B1

```



■ 12.4 MOVEM 命令実行例

【例2】 ポストインクリメント付レジスタ間接アドレス形式を用いると、アドレスレジスタ A0 には、命令実行の途中で一旦、\$FFFA0A1 の値が記憶されるが、命令終了時には、最終実効アドレス \$00001008 が記憶される。

```

MOVEA .L    # $1000, A0
MOVEM .W    (A0)+, D0/D3/A0-A1
ORG         $1000
DC .L       $80819091
DC .L       $A0A1B0B1

```

データ転送後のレジスタには以下の値がセットされる。

```

(D0)=$FFFF8081
(D3)=$FFFF9091
(A0)=$00001008
(A1)=$FFFFB0B1

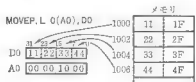
```

d. MOVEP (Move Peripheral Data) 命令<sup>†</sup> MOVEP 命令は、データレジスタとメモリの間で、バイト単位でデータ転送を行う。指定できるオペランドサイズはワードおよび

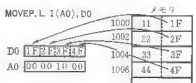
<sup>†</sup> MOVEP Dx, d(Ay) データレジスタ Dx → d(Ay)  
 MOVEP d(Ay), Dx d(Ay) の内容 → データレジスタ Dx

びロングワードであるので、バイトデータを数回に分けて転送する。すなわち、ワードの場合は2回、ロングワードの場合は4回である。

MOVEP 命令におけるメモリ実効アドレスはディスプレースメント付アドレスレジスタ間接形式のみ指定できる。データ転送は、データレジスタに対しては上位バイトから順に、メモリに対しては上記アドレス形式で指定される先頭アドレスから2番地ずつ増加するような順序で行われる。本命令ではバイトデータの転送を行うために、オペランドサイズがワードおよびロングワードであるにもかかわらず、メモリのデータ転送開始アドレスは、奇数および偶数のいずれでもよい。本命令は6800の周辺LSIのような8ビットデー



(a) 偶数アドレスの場合



(b) 奇数アドレスの場合

図 12.5 MOVEP 命令実行例

の場合は、\$1000と\$1002番地の2バイトがD0の下位に格納される。

```
MOVEA .L    # $1000, A0
MOVEP .L    0(A0), D0
```

(ii) 奇数アドレスの場合(図12.5(b)) この例では実効アドレスの先頭は\$1001番地である。メモリの奇数番地の内容(各ワーパの下位バイト)が4回に分けて、データレジスタD0の32ビットに転送される。なおオペランドサイズがワードの場合は、\$1001と\$1003番地の2バイトがD0の下位ワードに格納される。

```
MOVEA .L    # $1000, A0
MOVEP .L    1(A0), D0
```

e. MOVEQ (Move Quick) 命令 MOVEQ 命令は、図12.6に示すようにオペレーションワードのなかに含まれる8ビットのイミディエイトデータをデータレジスタへ転

↑ MOVEQ &(データ), Dn &(データ)→Dn

ータバスに接続されるデバイスとのデータ転送に有効である。

図12.5に示すように、メモリのアドレス指定が、偶数と奇数の場合のプログラム例を以下に示す。

#### (i) 偶数アドレスの場合(図12.5(a))

この例では実効アドレスの先頭は\$1000番地である。メモリの偶数アドレスの内容(各ワードの上位バイト)が4回に分けて、データレジスタD0の32ビットに転送される。なおオペランドサイズがワードの場合



送するものである。このとき 32 ビットに符号拡張してデータレジスタへ転送する。イミディエイトデータは符号付の 8 ビットデータとして表現されるため、10 進数では  $-128 \sim +127$  の範囲の値を取り得る。

MOVE および MOVEQ のイミディエイトアドレス形式の実行の比較を以下に示す。なお \* はこの命令によって値が変化しない部分を示す。

MOVE .L	##\$ 80, D0	(D0)=\$0 0 0 0 0 0 8 0; 符号拡張なし
MOVE .B	##\$ 80, D1	(D1)=\$* * * * * 8 0; 符号拡張なし
MOVEQ	##\$ 80, D2	(D2)=\$F F F F F F 8 0; 符号拡張あり
MOVE .L	##-128, D3	(D3)=\$F F F F F F 8 0; 符号拡張なし
MOVE .B	##-128, D4	(D4)=\$* * * * * 8 0; 符号拡張なし
MOVEQ	##-128, D5	(D5)=\$F F F F F F 8 0; 符号拡張あり

アセンブラでは、MOVE .L 命令でイミディエイト値が  $-128 \sim 127$  の範囲であると、MOVE .L 命令の代りに MOVEQ 命令のオペレーションコードを割り当てる。すなわち今述べた例では 4 番目と最後の命令の転送データは等しい。MOVE .L 命令と MOVEQ 命令を比較すると、バイト数と実行サイクル数がそれぞれ、6 バイトと 12 クロックサイクル、2 バイトと 6 クロックサイクルであるため、MOVEQ 命令の方がバイト数、サイクル数ともに有利であることがわかる。なお MOVEQ 命令は MOVE 命令と同様、コンディションコードレジスタに影響を与える。

f. LEA (Load Effective Address) 命令† LEA は、ソースオペランドの実効アドレス値を計算してその結果をデスティネーションオペランドで指定されるアドレスレジスタにセットするものである。デスティネーションのアドレスレジスタの 32 ビットはすべて影響を受ける。次に命令の使用例 (1), (2), (3) を示す。

(1) 本命令の先頭アドレスから 16 バイト増加したアドレス値を A0 にセットする。なお、この命令自身が 4 バイト長である。

```
LEA    *+$ 10, A0
```

(2) A1 の内容に +2 して、A1 に再格納する。

```
LEA    2(A1), A1
```

† LEA <ea>, An    ea で表されるアドレス → An

- (3) 16進の1000番地をアドレスレジスタA3にセットする。

LEA \$1000, A3

g. PEA (Push Effective Address) 命令<sup>f1</sup> PEA 命令は、デスティネーションオペランドの実効アドレスを計算して求め、その結果をシステムスタックに格納する。転送されるアドレスデータは必ずロングワードである。例を次に示す。

- (1) アドレスレジスタA5の内容をスタックに退避する。

PEA (A5)

- (2) アドレスレジスタA6の内容を+2して、スタックに退避する。

PEA 2(A6)

- (3) 16進の1000番地をアドレスデータとしてスタックに退避する。

PEA \$1000

h. EXG (Exchange Registers) 命令<sup>f2</sup> EXG 命令は、次のように2つのレジスタ間で、その内容を相互に交換する。

- (1) データレジスタ — データレジスタ (図12.7参照)  
 (2) アドレスレジスタ — アドレスレジスタ  
 (3) データレジスタ — アドレスレジスタ



■ 12.7 EXG 命令 (レジスタ間のデータ交換を行う)

本命令のオペランドサイズはロングワードのみである。

2つのレジスタ間で内容の交換を行う場合、本命令を用いると、命令バイト数、処理時間、一時記憶領域が不要などの面から有利となる。

i. SWAP (Swap Register Halves) 命令<sup>f3</sup> SWAP 命令は図12.8に示すように、指定されたデータレジスタの上位ワード (16ビット)

と下位ワード (16ビット) の内容を入れ替える。コンディションコードレジスタは MOVE 命令と同様に演算結果によって影響を受ける。割算を例として SWAP

命令の使い方を次に示す。

DIVS 命令を実行すると、データレジスタの下位ワ



図 12.8 SWAP 命令 (同一レジスタ内の上位ワードと下位ワードのデータ交換を行う)

- f1 PEA <ea> ea で表されるアドレス → システムスタック  
 f2 EXG Rx, Ry Rx と Ry の内容を交換する。  
 f3 SWAP Dn Dn の上位ワードと下位ワードを入れ替える。



ードに商、上位ワードに余りが格納される。これを分離して別のメモリ番地（商のアドレス=QUONT、余りのアドレス=RMNDR）へ移す。

MOVEQ	#-13, D2	(D2)=-13
DIVS	#5, D2	-13÷5
MOVE.W	D2, QUONT	商を QUONT へ格納する
SWAP	D2	D2 の上位, 下位ワードの入替えを行う
MOVE.W	D2, RMNDR	余りを RMNDR へ格納する

## 12.2 データ転送命令の応用例

### 12.2.1 単純なデータ転送

ここでは単純なデータ転送の例を示す。オペランドサイズとしてバイト、ワード、ロングワードのものを使用した。また、アドレス形式としては、データレジスタ直接形式、アドレスレジスタ直接形式、アドレスレジスタ間接形式、ディスプレースメント付/インデックス付アドレスレジスタ間接形式、イミディエイトデータ形式を適宜組み合わせた。データレジスタ、アドレスレジスタ、メモリの間で、種々のサイズのデータを、いろいろなアドレス形式を使用して転送している。

	ORG	\$1000	開始アドレス=\$1000
	MOVE.B	DATA1, D0	\$04→D0
	MOVE.W	DATA2, D1	\$1020→D1
	MOVE.L	DATA3, D2	\$30405060→D2
	MOVEA.W	#\$10FF, A0	\$000010FF→A0
	MOVE.B	D0, (A0)	\$04→\$10FF 番地
	MOVE.W	D1, 1(A0)	\$10→\$1100 番地
			\$20→\$1101 番地
	MOVE.L	D2, -1(A0, D0)	\$30→\$1102 番地
			\$40→\$1103 番地
			\$50→\$1104 番地
			\$60→\$1105 番地
DATA 1	DC.B	\$4	DATA1 の定数定義
DATA 2	DC.W	\$1020	DATA2 の定数定義
DATA 3	DC.L	\$30405060	DATA3 の定数定義

### 12.2.2 ブロック転送

まとまった量のデータ転送をブロック転送とよぶことが多い。68000 では 16M バイトまでのブロック転送を数命令で実行できる。しかし実際には、大量のデータを転送する場合

合には、より高速転送のできるダイレクトメモリアクセス法(DMA)を利用する場合は多い。したがって命令によるブロック転送は、比較的少量のデータ転送に適している。以下にブロック転送プログラムの例を示す。

	MOVEA .L	SPOINT, A0	ソースのポインタを設定する
	MOVEA .L	DPOINT, A1	デスティネーションポインタを設定する
	MOVE .L	COUNT, D0	転送語数を設定する
LOOP	MOVE .W	(A0)+, (A1)+	データ転送を行う
	DBRA	D0, LOOP	D0を1減ずる。結果が-1でないとLOOPへ分岐する
	SPOINT DC .L	\$10000	ソースポインタのスタートアドレス
	DPOINT DC .L	\$20000	デスティネーションポインタのスタートアドレス
	COUNT DC .L	1000	転送語数

この例では、\$10000番地から始まる1001語のデータブロックを、\$20000番地から始まる領域へ転送する。MOVE命令の両オペランドにはポストインクリメント付アドレスレジスタ間接形式を用いている。このプログラムは、22バイトのプログラムサイズである。また、1ワードの転送時間は23クロックサイクルである。68000のクロック周波数を8MHzとした場合、このワード転送に要する時間は2.875 $\mu$ sとなる。なお、転送語数が偶数ワードの場合には前の例のLOOPとラベル付けされたMOVE命令のオペランドサイズをロングワードにすると、ブロック転送をより速く実行できる。

### 12.2.3 プログラムスタック (Program STACK)

68000には、2章で述べたように、システムスタックとしてスーパーバイザシステムスタックとユーザシステムスタックがある。これらのスタックポインタとして、アドレスレジスタA7を割り当てている。本節ではこのようなシステムスタックではなく、プログラムの中で必要に応じて作ることができるプログラムスタックの構成について説明する。プログラムスタックを実現するためには、アドレスレジスタA0からA6の7本のアドレスレジスタのうちの1つを使用してポストインクリメント、あるいはプリデクリメントアドレスレジスタ間接のアドレス形式を伴うデータ転送命令を用いる。メモリ番地が減少する方向にプログラムスタックを構成する場合、プッシュ(データの格納)としてプリデクリメントアドレスレジスタ間接形式のMOVE命令を、ポップ(データの再読出し)にはポストインクリメントアドレスレジスタ間接形式のMOVE命令を用いる。逆にメ

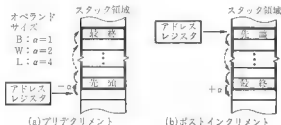


図 12.9 スタック操作上の注意

メモリ番地が増加する方向にプログラムスタックを構成する場合には MOVE 命令のアドレス形式としてプッシュにはポストインクリメント間接を、ポップにはプリデクリメント間接を用いる。

この操作を図 12.9 を用いて、以下に説明する。

(1) プリデクリメント アドレス レジスタ間接形式の MOVE 命令において、68000 はアドレスレジスタを  $\alpha$  の値だけデクリメントした後で、そのレジスタをスタックのポインタとして用いる。したがってアドレスレジスタにはスタックの先頭番地  $+\alpha$  の値を初期設定しておく必要がある。ここで  $\alpha$  はオペランドサイズがバイトのとき 1、ワードのとき 2、ロングワードのとき 4 である。

(2) ポストインクリメント アドレス レジスタ間接形式の場合、68000 はアドレスレジスタをプログラムスタックのポインタとして用いた後、インクリメントする。したがって命令実行後、レジスタの値には、 $\alpha$  ( $\alpha$  の値は (1) と同様) が加算される。

(3) バイトデータとワードデータ (ロングワードデータも含む) とをプログラムスタック内に混在させて記憶する場合には、後にプログラムスタックからそのデータを再び読み出すときに配列誤りを起さないように、プログラム作成上の注意が必要である。すなわち、オペランドサイズがワードやロングワードの場合には、実効アドレスが常に偶数になるようにしなければアドレスエラーが生ずる。したがってバイトデータを扱う場合には、ダミーのバイトデータを本来のバイトデータに付加し、ワードデータのようにして扱うような配慮も必要となる。

次にスタック形記憶の例として以下に数値計算の一方法を示す。

$$X = A + B \times (C + D)$$

これを逆ポーランド記法を用いて書き換えると次のようになる。

$$X = ABCD + \times +$$

メモリ番地が小さくなる方向に構成されたプログラムスタックを用いて、上式を実行するプログラムを作成すると次のようになる。

MOVE .L	#STKP+2, A0	スタック ポインタ A0 を初期設定する
MOVE .W	INPUT, -(A0)	A をプログラム スタックに格納する
MOVE .W	INPUT, -(A0)	B をプログラム スタックに格納する
MOVE .W	INPUT, -(A0)	C をプログラム スタックに格納する
MOVE .W	INPUT, D0	D を D0 に格納する
ADD .W	(A0)+, D0	C+D
MULS .W	(A0)+, D0	B(C+D)
ADD .W	(A0)+, D0	A+B(C+D)

この例では、外部入力装置 (INPUT 番地に割り付けられている) から 16 ビットのデータを順次入力する。68000 の入力動作に同期して外部入力機器は 680000 への出力動作をしていると仮定している。このプログラムでは数値の入力順にプログラムスタックでこれを記憶していくため、データレジスタを用いないですむことや、アドレス形式に A0 修飾を用いているので命令バイト数が少なくなるなどの特徴をもつ。68000 はスタックに対して種々演算を行えるため、いわゆるスタックマシンと同様の機能が実現できる。

#### 12.2.4 キュー (QUEUE)

キューはスタックと同様に、アドレスを示すポインタを更新しながらデータを格納したり読み出したりするものである。しかしスタックが、あとから格納したデータから順に読み出す (Last-in-First-out=LIFO) のに対してキューではさきに格納したデータから順に読み出す (First-in-First-out=FIFO) ことが特徴となっている。このようにキューでは格納した順序と同じ順序でデータを読み出すため、データ格納用ポインタとデータ読出し用ポインタの対が必要である。このように、プログラムでキューを構成するために、A0 から A6 までのアドレスレジスタのうち 2 本を使用する。メモリ番地が増加する方向にキューを構成する場合には、データ格納とデータ読出しを行う MOVE 命令のアドレス形式として、共にポストインクリメントアドレスレジスタ間接形式を用いる。逆にメモリ番地が減少する方向にキューを構成する場合には、共にプリデクリメントアドレスレジスタ間接のアドレス形式を用いる。

スタックは LIFO であるのでデータ格納とデータ読出しの頻度が同じ程度ならスタック領域に大きな増減はない。しかしキューは FIFO のため、データ格納が続くときキュー領域はそれに応じて増加し続ける。これを防ぐためには、次のようなキューの操作をプログ

ラムで工夫する必要がある。プログラムでキューの容量を指定して、データ格納用ポインタの内容がこの指定値を超えたら、データ格納用ポインタを初期値（キューの格納開始番地）に再設定する。同様に、データ読出しポインタの内容がキュー容量指定値を超えたら初期設定を行う。このようなデータ格納用ポインタとデータ読出し用ポインタを初期値に再設定しながら用いるキューを循環バッファとよぶ。循環バッファを構成する場合、データ格納用ポインタの内容とデータ読出しポインタの内容の差を常に監視して、いわゆるオーバーランが生じないようなプログラム上の注意が必要である。

# 13 算術論理演算命令

この章では、各算術論理演算命令についてその機能を説明する。各命令のソースオペランドおよびデスティネーションオペランドに許されるアドレス形式については、付録の表を参照されたい。また各命令のコンディションコードの変化については6章を参照されたい。さらにオペランドサイズに関しては通常の命令ではバイト、ワード、ロングワードの3種類が使用可能である。特に制限がある場合のみ説明を加える。

## 13.1 算術演算命令

算術演算命令には加算、減算、乗算、除算のほか補数、比較、テスト、エクステンド、クリア命令がある。以下各命令毎に説明する。

### 13.1.1 加算命令

加算命令には ADD, ADDA, ADDI, ADDQ, ADDX 命令の5種の命令がある。このうち ADDX は多倍長加算命令であり、13.1.4 項で説明する。

**a. ADD (Add Binary) 命令†** この命令はデータの加算に使用する命令であり、デスティネーションオペランドの内容にソースオペランドの内容を加算し、結果をデスティネーションに格納する。

デスティネーションオペランドにアドレスレジスタを指定すると、命令の機能がコンディションコードの変化も含めて ADDA 命令に一致する。

---

†  $\text{ADD } \langle \text{ea} \rangle, \text{Dn} \quad \text{Dn} + \langle \text{ea} \rangle \longrightarrow \text{Dn}$   
 $\text{ADD } \text{Dn}, \langle \text{ea} \rangle \quad \langle \text{ea} \rangle + \text{Dn} \longrightarrow \langle \text{ea} \rangle$

**b. ADDA (Add Address) 命令<sup>†1</sup>** この命令はアドレス データの加算に使用する命令である。デスティネーション オペランドで指定したアドレス レジスタ  $An$  にソース オペランドの内容を加算し、結果をアドレス レジスタ  $An$  に格納する。オペランドサイズはワードおよびロングワードに制限されておりバイトは指定できない。ワード長の演算が指定された場合にはソース オペランドの符号ビット(ビット 16)を拡張してロングワードデータとして加算を行う。

ADDA 命令はアドレス データを加算するための命令なので、コンディション コードは変化しない。

**c. ADDI (Add Immediate) 命令<sup>†2</sup>** この命令はデスティネーション オペランドの内容にイミディエイト データを加算し、結果をデスティネーションに格納する命令である。オブジェクト コードではイミディエイト データが命令ワードの次に付加される。デスティネーション オペランドにアドレス レジスタを指定することはできない。アドレス レジスタにイミディエイト データを加算するには ADDA 命令を用いる。

**d. ADDQ (Add Quick) 命令<sup>†3</sup>** この命令は ADDI 命令と同様にイミディエイト データの加算をする命令であり、データに対する加算とアドレス データに対する加算のどちらにも使うことができる。イミディエイト データの範囲は 1 から 8 までは限定されているが、この範囲のデータなら ADDI 命令より ADDQ 命令のほうが命令バイト数が少なく、命令実行時間が短い。たとえばデータ レジスタへ加算する場合の命令バイト数と命令クロック サイクル数を比較すると次のようになる。

	命令長	実行時間
ADDQ.W #2, D0	2 バイト	4 クロック サイクル
ADDI.W #2, D0	4 バイト	8 クロック サイクル

クイック命令ではイミディエイト データが命令ワード中の 3 ビットで示されるために命令バイト数が少ない。またイミディエイト命令と異なりイミディエイト データを読み出すサイクルが不要なため実行時間が短い。

オペランドサイズはアドレス レジスタを使うときにのみワードまたはロングワードに制限される。

<sup>†1</sup> ADDA <ea>,  $An$        $An + \langle ea \rangle \rightarrow An$

<sup>†2</sup> ADDI #<データ>, <ea>       $\langle ea \rangle + \# \langle \text{データ} \rangle \rightarrow \langle ea \rangle$

<sup>†3</sup> ADDQ #<データ>, <ea>       $\langle ea \rangle + \# \langle \text{データ} \rangle \rightarrow \langle ea \rangle$

## 13.1.2 減 算 命 令

減算命令には SUB, SUBA, SUBI, SUBQ, SUBX 命令の5種の命令がある。これらの命令のアドレス形式、コンディションコード、オペランドサイズは、それぞれ対応する加算命令の場合と一致する。

なお SUBX 命令は 13.1.4 項で説明する。

a. SUB (Subtract Binary) 命令<sup>†1</sup> この命令はデータの減算に使用する命令である。デスティネーションオペランドの内容からソースオペランドの内容を減算し、結果をデスティネーションに格納する。

デスティネーションオペランドにアドレスレジスタを指定すると命令の機能がコンディションコードの変化も含めて SUBA 命令に一致する。

SUB 命令を実行すると、演算結果によりコンディションコードが設定される。C フラグは Borrow が発生するとセットされ、それ以外のときはクリアされる。他のフラグの変化は ADD 命令と等しい。

b. SUBA (Subtract Address) 命令<sup>†2</sup> この命令はアドレスデータに対する減算に使用する命令である。デスティネーションオペランドで指定したアドレスレジスタ An の内容からソースオペランドの内容を減算し、結果をアドレスレジスタ An に格納する。オペランドサイズはワードおよびロングワードに制限されておりバイトは指定できない。ワード長の演算が指定された場合にはソースオペランドの符号ビット (ビット 16) を拡張してロングワードデータとして減算を行う。

SUBA 命令ではコンディションコードは変化しない。

c. SUBI (Subtract Immediate) 命令<sup>†3</sup> この命令はデスティネーションオペランドの内容からイミディエイトデータを減算し、結果をデスティネーションに格納する命令である。オブジェクトコードではイミディエイトデータが命令ワードの次に付加される。デスティネーションオペランドにアドレスレジスタを指定することはできない。アドレスレジスタからイミディエイトデータを減算するには SUBA 命令を用いる。

d. SUBQ (Subtract Quick) 命令<sup>†4</sup> この命令は SUBI 命令と同様にイミディエイト

†1 SUB <ea>, Dn  $Dn - \langle ea \rangle \rightarrow Dn$

SUB Dn, <ea>  $\langle ea \rangle - Dn \rightarrow \langle ea \rangle$

†2 SUBA <ea>, An  $An - \langle ea \rangle \rightarrow An$

†3 SUBI <データ>, <ea>  $\langle ea \rangle - \# \langle \text{データ} \rangle \rightarrow \langle ea \rangle$

†4 SUBQ <データ>, <ea>  $\langle ea \rangle - \# \langle \text{データ} \rangle \rightarrow \langle ea \rangle$



データを減算する命令であり、データに対する減算とアドレスデータに対する減算のどちらにも使うことができる。イミディエイトデータの範囲は1から8までに限定されているが、この範囲のデータなら SUBI 命令より SUBQ 命令のほうが命令バイト数が少なく、命令実行時間が短い。アドレスレジスタを使うときオペランドサイズはワードまたはロングワードに制限される。

### 13.1.3 補数命令

補数命令には NEG、NEGX 命令の2種の命令がある。NEGX 命令は次の多倍精度演算の項で説明する。

**NEG (Negate Binary) 命令<sup>t1</sup>** この命令はデスティネーションオペランドの内容をゼロから減算し、デスティネーションへ格納する命令である。

コンディションコードの設定は SUB 命令と同様に行われる。

### 13.1.4 多倍精度演算命令

68000 ではデータレジスタが32ビット長であるので、33ビット以上のデータを表すには、いくつかのレジスタまたはメモリを使ってデータを表すことになる。この多倍精度の演算を行うための命令が ADDX、SUBX、NEGX である。

**a. ADDX (Add Multi-precision) 命令<sup>t2</sup>** この命令はデスティネーションオペランドの内容とソースオペランドの内容と X フラグを加算してデスティネーションへ格納する命令である。脚注に示すようにデータレジスタ間の加算およびメモリ間(ブリダトリメントアドレスレジスタ間接形式で指定)の加算ができる。

多倍精度加算の原理を図 13.1 に示す。下位の桁どうしから加算を始め、上位の桁へのキャリイは、その有無を X フラグ<sup>n</sup>(2 ページ)に記憶しておき、上位桁の加算のときに X



図 13.1 多倍精度加算

- |    |      |              |   |
|----|------|--------------|---|
| t1 | NEG  | <ea>         | $0 - \langle ea \rangle \rightarrow \langle ea \rangle$ |
| t2 | ADDX | Dy, Dx       | $Dx + Dy + X \text{ フラグ} \rightarrow Dx$                |
|    | ADDX | -(Ay), -(Ax) | $(Ax) + (Ay) + X \text{ フラグ} \rightarrow (Ax)$          |

フラグの内容も含めて加算することにより実現する。

コンディション コードは Z フラグを除いて通常の ADD 命令と同様に变化する<sup>11</sup>。Z フラグは演算結果が非ゼロの場合クリアされ、ゼロの場合は变化しない。この機能を使えば多倍精度数全体の演算結果がゼロであるかないかの判定に Z フラグを使うことができる。この方法を図 13.2 に示す。まず多倍精度演算ループに入る前に Z フラグを立てておく。もし演算ループの途中で演算結果が非ゼロになると、Z フラグがクリアされるので、ループを抜けたときに多倍精度数がゼロでないことがわかる。また演算ループの途中の演算結果がすべてゼロのときには、Z フラグが立ったままループを抜けるので、多倍精度数がゼロであることがわかる。

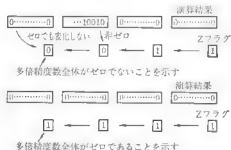


図 13.2 多倍精度演算での Z フラグの使い方

次に多倍精度加算のプログラム例を示す。図 13.2 に示すように 2 つの 4 ワード長データがそれぞれメモリに格納されている。アドレスレジスタ A0 および A1 が各先頭アドレスを指して示しているものとする。プログラムではまず A0, A1 の値を 8 増して下位桁を指し示すようにする。4 ワードの加算ができるようにループカウンタ用のデータレジスタ D0 に 3 をセットする。MOVE # \$04, CCR 命令で Z フラグをセットし他のフラグをクリアする。次に加算ループに入るが、このループは X フラグに変化を与えない DBRA 命令を用いて作る。

ADDQ .L	#8, A0	下位桁のポインタアドレスをセットする。
ADDQ .L	#8, A1	
MOVEQ .L	#3, D0	ループカウンタセット
MOVE	# \$04, CCR	X フラグクリア
L01	ADDX	-(A0), -(A1) 拡張加算
	DBRA	D0, L01

このループを抜けたとき、X, Z, N, V の各フラグはそれぞれ、多倍精度数全体に対す

<sup>11</sup> (前ページ) 多倍精度演算命令では桁上げ情報を X フラグに記憶させている。このため多倍精度演算ループではループ中の他の命令が X フラグを変化させてはいけない。DBcc 命令や CLR 命令などは、X フラグを変化させないため、ループの中に含めてよい。C フラグは多くの命令で変化するため桁上げ情報の保持には適さない。

<sup>12</sup> 6 章参照

るキャリイ、ゼロ、負、オーバフローを示している。

**b. SUBX (Subtract Multi-precision) 命令<sup>†1</sup>** この命令は、デスティネーションオペランドの内容からソースオペランドの内容と X フラグを減算して、デスティネーションに格納する命令である。データレジスタ間の減算およびメモリ間の減算ができる。

多倍精度数の減算は、下位の桁からの Borrow を X フラグに記憶しておき、次の桁の減算で X フラグを含めて減算するとにより実現される。

**c. NEGX (Negate Multi-precision) 命令<sup>†2</sup>** この命令は、0 からデスティネーションオペランドの内容と X フラグの内容を減算して、デスティネーションに格納する命令である。この命令を使って多倍精度数全体の補数をとることができる。コンディションコードは SUBX 命令と同様な変化をする。

### 13.1.5 クリア命令

**CLR (Clear Operand) 命令<sup>†3</sup>** この命令は、デスティネーションオペランドの全ビットをゼロにクリアする命令である。コンディションコードに関しては、X フラグが変化せず、Z フラグがセットされ、他のフラグはクリアされる。

### 13.1.6 符号拡張命令

**EXT (Extend) 命令<sup>†4</sup>** データレジスタの符号ビットにあたるビットを上位方向にコピーすることにより、バイトデータをワードデータへ、またはワードデータをロングワードデータへ変換する命令である。この様子を図 13.3 に示す。オペランドサイズがワードの場合、データレジスタの



■ 13.3 符号拡張命令の操作

ビット 7 を 8~15 ビットに拡張し、オペランドサイズがロングワードの場合、

ビット 15 を 16~31 ビットに拡張する。コンディションコードの X フラグは変化せず、V フラグ、C フラグはゼロにリセットされ、N フラグ、Z フラグは演算結果によりセットされる。

†1 SUBX	Dy, Dx	$Dx - Dy - X \text{ フラグ} \rightarrow Dx$
	SUBX $-(Ay), -(Ax)$	$(Ax) - (Ay) - X \text{ フラグ} \rightarrow (Ax)$
†2 NEGX	<ea>	$0 - \langle ea \rangle - X \text{ フラグ} \rightarrow \langle ea \rangle$
		$0 \rightarrow \langle ea \rangle$
†3 CLR	<ea>	
†4 EXT	Dn	ワードのとき 7 ビット目 $\rightarrow$ 8~15 ビット, ロングワードのとき 15 ビット目 $\rightarrow$ 16~31 ビット

## 13.1.7 テスト命令

**TST (Test) 命令<sup>†1</sup>** この命令は、デスティネーションオペランドが負であるかゼロであるかをコンディションコードの N フラグまたは Z フラグへセットする命令である。X フラグは影響されず、V フラグと C フラグはゼロにリセットされる。テスト命令は、オペランドが正、負、ゼロのいずれかによってブランチするための条件づくりに使われる。

## 13.1.8 比較命令

比較命令は、デスティネーションオペランドの内容からソースオペランドの内容を減算し、コンディションコードをセットする命令である。各オペランドの内容は比較命令の実行後にも変化しない。通常、比較命令のあとにブランチ命令を置き条件ブランチを実現する。

68000 の比較命令には CMP, CMPA, CMPI, CMFM 命令の 4 種類がある。

**a. CMP (Compare) 命令<sup>†2</sup>** この命令は、デスティネーションオペランドで指定したデータレジスタ Dn の内容から実効アドレスで指定したソースオペランドの内容を減算し、その結果に従ってコンディションコードを設定する命令である。データレジスタの内容は変化しない。実効アドレスにはすべてのアドレス形式が可能であるが、アドレスレジスタを指定するときにはバイトサイズのエレメンションはできない。

次に CMP 命令を用いた最大値検索のプログラム■を示す。図 13.4 に示すようにアドレスレジスタ A0 で指し示されるメモリ領域に、検索用データが収納されているテーブルがあるとする。データはワード長のゼロでない符号なし数で、テーブルの終わりにはゼロが入っていると。初期設定としてレジスタ D0 をクリアする。まずテーブル内のデータをレジスタ D1 に取り出す。取り出した内容がゼロであればテーブルの終りなのでこのルーチンを終了する。ゼロでなければデータレジスタ D1, D0 の内容を比較し大きい方の値を D0 にセットして次の比較へす

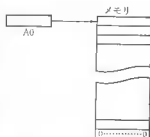


図 13.4 最大値検索プログラムのデータ

†1 TST <ea>      0-<ea> → フラグへ反映  
 †2 CMP <ea>, Dn    Dn-<ea> → フラグへ反映

む。このルーチンを終了したときには、データレジスタ D0 に最大値を得る。

	CLR	D0	データレジスタのクリア
L01	MOVE	(A0)+, D1	データを D1 へ取り出す
	BEQ	NEXT	テーブルの終りであれば NEXT へ
	CMP	D1, D0	比較
	BHI	L01	小さければそのまま次のデータへ
	MOVE	D1, D0	大きければ D0 へセット
	BRA	L01	次のデータへ
NEXT	EQU	*	NEXT STEP

b. CMPA (Compare Address) 命令<sup>11</sup> この命令は、デスティネーションオペランドで指定したアドレスレジスタ An の内容からソースオペランドの内容を減算し、結果に従ってコンディションコードを設定する命令である。アドレスレジスタの内容は変化しない。オペランドサイズはワード、ロングワードの2種類が指定でき。ソースオペランドの実効アドレスにはすべてのアドレス形式が指定できる。

c. CMPI (Compare Immediate) 命令<sup>12</sup> この命令は、デスティネーションオペランドの内容からイミディエイトデータを減算し、結果に従ってコンディションコードを設定する命令である。デスティネーションオペランドの内容は変化しない。

d. CMPM (Compare Memory) 命令<sup>13</sup> この命令は、デスティネーションオペランドの内容からソースオペランドの内容を減算し、結果に従ってコンディションコードを設定する命令である。両オペランドの内容は変化しない。どちらのオペランドともポストインクリメントアドレスレジスタ間接形式でアドレッシングされるので、メモリの内容を順番に比較する場合に便利な命令である。

### 13.1.9 乗算命令

乗算命令には符号付および符号なしの乗算命令がある。乗算命令は、データレジスタの下位 16 ビットのデータとソースオペランド 16 ビットのデータとを乗算し 32 ビットの積をデータレジスタに格納する命令である。

a. MULU (Unsigned Multiply) 命令<sup>14</sup> この命令は 2 つの符号なし 16 ビット整数を乗算し、符号なし 32 ビット整数を求める命令である。ソースオペランドの実効アドレ

<sup>11</sup> CMPA <ea>, An An-<ea> → フラグへ反映

<sup>12</sup> CMPI #<データ>, <ea> <ea>-#<データ> → フラグへ反映

<sup>13</sup> CMPM Ay@+, Ax@+ (Ax)-(Ay) → フラグへ反映

<sup>14</sup> MULU <ea>, Dn Dn×<ea> → Dn

スは、アドレスレジスタ直接形式を除いて他のすべてのアドレス形式で指定することができる。特にデータレジスタを指定するときにはそのレジスタの下位16ビットのみが取り出され、上位16ビットは無視される。デスティネーションオペランドに指定したデータレジスタには32ビットが全部格納される。

演算結果によって、コンディションコードのZフラグとNフラグがセットまたはクリアされる。2つの16ビットの積を32ビットで表すので、オーバフローが起きることはなくVフラグおよびCフラグはクリアされる。またXフラグは影響されない。

次に2つの符号付32ビット整数を乗算し、符号付64ビット整数を求めるサブルーチンの例を示す。このプログラムは、データレジスタD0およびD1にある2つの符号付32ビット整数の積を求め、下位32ビットをD0に、上位32ビットをD1に出力するルーチンである。MULU命令では16ビット整数どうしの積しか求められない。そこで図13.5に示すように部分積を求め、後に加算する方法をとる。プログラムではSTEP0で

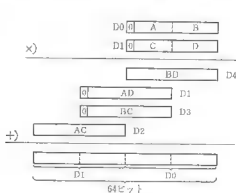


図 13.5 32ビット乗算の部分積

2つの数の符号を調べ、正の数に直し、積の符号をD7に記憶する。STEP1では部分積を求めるためデータを図13.6のようにレジスタに設定する。STEP2で各部分積を求める。部分積ADとBCのMSBはゼロなので、STEP3で和を求めたときキャリイは発生しない。STEP4で64ビットの積を求め、STEP5で符号をセットする。

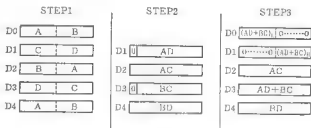


図 13.6 各ステップでのレジスタ内容

MUL 64	EQU	*	
	CLR. W	D7	
STEP 0	TST. L	D0	D0 データの符号を調べる
	BPL. S	SIGN 2	
	NEG. L	D0	負なら正に直す
	NOT. W	D7	
SIGN 2	TST. L	D1	D1 データの符号を調べる
	BPL. S	STEP 1	
	NEG. L	D1	負なら正に直す
	NOT. W	D7	積の符号を D7 に記憶する
STEP 1	MOVE. L	D0, D2	
	MOVE. L	D1, D3	
	SWAP	D2	
	SWAP	D3	
	MOVE. L	D0, D4	
STEP 2	MULU	D1, D4	部分積 BD
	MULU	D2, D1	部分積 AD
	MULU	D3, D2	部分積 AC
	MULU	D0, D3	部分積 BC
STEP 3	ADD. L	D1, D3	AD+BC
	CLR. L	D0	
	CLR. L	D1	
	MOVE. W	D3, D0	
	SWAP	D0	(AD+BC) <sub>L</sub> , 0
	SWAP	D3	
	MOVE. W	D3, D1	0, (AD+BC) <sub>H</sub>
STEP 4	ADD. L	D4, D0	下位 32 ビット
	ADDX. L	D2, D1	上位 32 ビット
STEP 5	TST. W	D7	
	BEQ. S	EXT	
	NEG. L	D0	負のとき補数に直す
	NEGX. L	D1	
EXT	RTS		
	END		

**b. Muls (Signed Multiply) 命令†** この命令は、2つの符号付 16 ビット整数を乗算し、符号付 32 ビット整数を求める命令である。演算は符号付の算術演算で行われる。オペランドのアドレス形式、扱うビット長およびコンディションコードの変化は、MULU 命令と等しい。

† Muls <ea>, Dn    Dn × <ea> → Dn

## 13.1.10 除 算 命 令

除算命令には符号付および符号なしの除算命令がある。除算命令はデータレジスタの32ビットの整数をソースオペランドの16ビット整数で除算し、商と余りをデータレジスタに格納する。図13.7に示すように結果の

32ビットの内容は次のとおりである。

(1) 商は下位ワード(下位16ビット)に格納されている。

(2) 余りは上位ワード(上位16ビット)に格納されている。

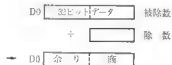


図 13.7 除算命令

符号付および符号なし除算命令はともに次の2つの特別な実行結果がある。

- (1) 0で割るとトラップが生じ、プロセッサは例外処理を開始する<sup>1)</sup>。
- (2) 命令完了前にオーバフローを検出すると、Vフラグがセットされ命令は実行されない。この場合データレジスタの値は更新されない。

**a. DIVU (Unsigned Divide) 命令<sup>1)</sup>** この命令はデータを符号なしの数とみなして除算する命令である。ソースオペランドはアドレスレジスタ直接形式を除いて、他のアドレス形式で指定することができる。

この除算命令はデータのビット長の関係からオーバフローが生じやすいので注意を要する。たとえばデータレジスタの32ビット目に1が立っているとき、これを1で割ると、商は32ビットデータとなり16ビットに収まりきらないためオーバフローとなる。そこで次のようなプログラムを使うことにより除算でのオーバフローを起きないようにすることができる。このプログラムでは32ビットのデータを上位16ビットと下位16ビットに分けて16ビットのデータで割って32ビットの商と16ビットの余りを得る。被除数はデータレジスタD0の32ビットに、除数はデータレジスタD2の下位16ビットに格納されているとする。まず被除数の下位16ビットをデータレジスタD1に移し、上位16ビットをレジスタD0の下位側へ移す。上位16ビットのデータをデータレジスタD2の除数で割ると、商がD0の下位へ、余りがD0の上位へ入る。この除算では、16ビットデータを16ビットデータで割ることになるので、オーバフローが発生することはない。次にこの余りとD1に格納された被除数の下位16ビットを加え、これを除数で割る。この除

<sup>1)</sup> 8章参照

<sup>2)</sup> DIVU <ea>, Dn    Dn+<ea> → Dn





図 13.8 オーバフローしない除算ルーチン

算でもオーバフローは発生しない。下位側の商をデータレジスタ D0 の下位 16 ビットに移せば、D0 全体で 32 ビットの商となる。このルーチンにおけるレジスタ内容の変化を図 13.8 に示す。

CLR	D1	
MOVE.W	D0, D1	被除数の下位 16 ビット
SWAP	D0	
ANDI.L	#\$FFFF, D0	被除数の上位 16 ビット
DIVU	D2, D0	除数で割る。
SWAP	D0	
SWAP	D1	
MOVE.W	D0, D1	余りを下位 16 ビットに移す
SWAP	D1	
DIVU	D2, D1	
MOVE.W	D1, D0	下位、上位の商を合成する

このルーチンを出るときに、余りはデータレジスタ D1 の上位 16 ビットに入っている。

**b. DIVS (Signed Divide) 命令†** この命令はデータを符号付の数とみなして除算する命令である。オペランドのアドレス形式、扱うビット長およびコンディションコードの変化は DIVU 命令と同じである。

余りが 0 でない場合、余りの符号は常に被除数と同じになる。たとえば、次のルーチンに示すように -12 を 5 で割ると、商は -2 となり、余りが -2 となる。

MOVE.L	#-12, D2	セットデータ
DIVS	#5, D2	-12/5
MOVE.W	D2, QUONT	-2 (商)
SWAP	D2	
MOVE.W	D2, RMNDR	-2 (余り)

† DIVS <ea>, Dn    Dn ← (Dn <ea>) → Dn

## 13.2 論理演算命令

68000 の論理演算命令には論理積、論理和、排他的論理和および否定命令がある。イミディエイト論理命令と否定命令を除いた通常の論理命令では、演算は必ずデータレジスタと他のオペランドとの間で行われる。ソースオペランドにデータレジスタを指定した場合デスティネーションオペランドにはアドレスレジスタ直接形式およびディスプレースメント付/インデックス付プログラムカウンタ相対形式を除いた、他のアドレス形式を用いることができる。またデスティネーションオペランドにデータレジスタを指定した場合には、ソースオペランドにアドレスレジスタ直接形式を除いた他のアドレス形式を用いることができる。アドレスレジスタ上のデータに論理演算を行うことはできない。

コンディションコードの変化は、イミディエイト論理命令でコンディションコードレジスタまたはステータスレジスタに直接論理演算を施す場合を除くと、どの論理演算でも共通で、6章の表6.2に示したようになる。この表によれば Z, N フラグが変化し、C, V フラグはクリアされる。また X フラグは前の値が保持される。

論理演算のオペランドサイズはバイト、ワード、ロングワードのすべてが可能である。イミディエイト論理演算命令 (ANDI, ORI, EORI) ではデスティネーションオペランドとしてコンディションコードレジスタまたはステータスレジスタを指定できるが、これらについては特権命令にもなるので、16, 17章にて改めて説明する。

**a. AND (Logical AND) 命令†** この命令は、デスティネーションオペランドの内容とソースオペランドの内容との論理積をとり、デスティネーションへ格納する命令である。

AND 命令の使用例としては、ビットデータの不要な部分にマスクをかけて消し、必要なビットのみを残す操作に用いる場合がある。たとえば ASCII コードから数値データを取り出す場合がそうである。数字の 0 から 9 を表す ASCII コードは 16 進数でそれぞれ "\$30", "\$31", ..., "\$39" である。そこでデータレジスタ D0 に数字コードが入っているとき下のプログラムのようにして下位 4 ビットのみ取り出せば数値データに変換できる。

```

AND.L    MASK, D0
        ⋮
MASK: DC.L    $000F      マスクデータ

```

---

† AND <ea>, Dn    Dn<ea> → Dn  
   AND Dn, <ea>    <ea>·Dn → <ea>

**b. ANDI (AND Immediate) 命令<sup>t1</sup>** この命令は、デスティネーションオペランドの内容とイミディエイトデータとの論理積をとり、デスティネーションへ格納する命令である。

AND 命令で示したプログラムを ANDI 命令で書き直すと

```
ANDI .L    # $OF, D0
```

となり、マスク情報が命令に含まれるためわかりやすくなる。

**c. OR (Inclusive OR logical) 命令<sup>t2</sup>** この命令は、デスティネーションオペランドの内容とソースオペランドの内容の論理和をとり、デスティネーションへ格納する命令である。

OR 命令はあるビット位置に無条件に1を立てるときや、データの一部に他のデータをはめこむときに用いることもできる。後者の一例として、ASCII 文字データで2進化10進数の2桁がデータレジスタ D0 と D1 に入っているとき、データレジスタ D0 に2進化10進数値をパックするルーチンは以下のようになる。

```
ANDI .L    # $OF, D0
LSL .W     # 4, D1
OR .B      D1, D0
```

**d. ORI (Inclusive OR Immediate) 命令<sup>t3</sup>** この命令は、デスティネーションオペランドの内容とイミディエイトデータとの論理和をとり、デスティネーションへ格納する命令である。

**e. EOR (Exclusive OR Logical) 命令<sup>t4</sup>** この命令は、デスティネーションオペランドの内容とソースオペランドの内容との排他的論理和をとり、デスティネーションへ格納する命令である。EOR 命令では AND 命令、OR 命令と異なり、ソースオペランドがデータレジスタのみに限定されている。

**f. EORI (Exclusive OR Immediate) 命令<sup>t5</sup>** この命令は、デスティネーションオペランドの内容とイミディエイトデータとの排他的論理和をとり、デスティネーションへ格納する命令である。

t1	ANDI	#〈データ〉, 〈ea〉	〈ea〉:#〈データ〉→〈ea〉
t2	OR	〈ea〉, Dn	Dn ∨ 〈ea〉 → Dn
	OR	Dn, 〈ea〉	〈ea〉 ∨ Dn → 〈ea〉
t3	ORI	#〈データ〉, 〈ea〉	〈ea〉 ∨ #〈データ〉 → 〈ea〉
t4	EOR	Dn, 〈ea〉	〈ea〉 ⊕ Dn → 〈ea〉
t5	EORI	#〈データ〉, 〈ea〉	〈ea〉 ⊕ #データ → 〈ea〉

**g. NOT (Logical Complement) 命令<sup>11</sup>** この命令は、デスティネーションオペランドの内容の1の補数を取り（各ビットを反転させ）、デスティネーションへ格納する命令である。

### 13.3 2進化10進数演算命令

68000には2進化10進数を演算する加算、減算および補数命令がある。これらの命令は1バイトにつめられた2進化10進数BCD (Binary Coded Decimal) の2桁を単位として演算を行う。そして多数桁の2進化10進数演算ができるように拡張フラグ(Xフラグ)を含めた演算を実行する。

**a. ABCD (Add Digits) 命令<sup>12</sup>** この命令は、Xフラグを含めてバイト長のオペランドどうしの2進化10進数加算を行う命令である。オペランドの指定のしかたは2とありの方法がある。

(1) データレジスタ間  $ABCD \quad Dy, Dx$

(2) メモリ間  $ABCD \quad -(Ay), -(Ax)$

メモリ間の演算では、どちらのオペランドもプリデクリメントアドレスレジスタ間接形式で指定する。またオペレーションサイズはバイト長だけが許されている。本命令ではコンディションコードのNフラグおよびVフラグは、意味が無いため定義されない。CフラグおよびXフラグは10進加算でキャリイが発生したときセットされ、その他のときクリアされる。Zフラグは演算結果がゼロでないときクリアされ、その他のときは変化しない。したがって多倍精度2進数のときと同じように、多バイトで表現された2進化10進数全体の演算結果がゼロであるかを判別することができる。

次に4バイト2進化10進数加算のプログラムを示す。図13.9に示すように、2つの4バイト2進化10進数がメモリに格納されており、ルーチンに入る前に、アドレスレジスタA0, A1がそれぞれのデータの先頭アドレスを指し示しているとする。このプログラムではまず各アドレスレジスタを4番地増加し最下位桁の1つ先のバイトを指し示させる。次に

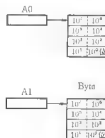


図 13.9 2進化10進数加算のデータ構造

<sup>11</sup> NOT  $\langle ea \rangle \quad \langle ea \rangle \rightarrow \langle ea \rangle$

<sup>12</sup> ABCD  $Dy, Dx \quad Dx + Dy + X \text{ フラグ} \rightarrow Dx$

ABCD  $-(Ay) -(Ax) \quad (Ax) + (Ay) + X \text{ フラグ} \rightarrow (Ax)$

ループカウンタに使うレジスタ D0 に 3 をセットし、X フラグをクリアし、Z フラグをセットする。この後 BCD 加算のループに入り 4 バイトの和データを得る。

```

ADDQ .L    #4, A0          下位バイト ポイント
ADDQ .L    #4, A1          下位バイト ポイント
MOVEQ .L   #3, D0         ループ カウンタ セット
MOVE      #$04, CCR        CCR クリア
L01:  ABCD  -(A0), -(A1)    10 進加算
      DBRA  D0, L01

```

このルーチンを出たあと、Z フラグは演算結果の 4 バイト 2 進化 10 進数がゼロであるかどうかを示しており、C フラグは 4 バイト 2 進化 10 進数の加算でのキャリイを示している。

**b. SBCD (Subtract Digits) 命令†** この命令は、X フラグを下位桁からのボローとして、2 進化 10 進数の減算を行う命令である。オペランドの指定の方法およびオペレーションサイズは ABCD 命令と変わらない。コンディションコードは、C フラグおよび X フラグが 10 進数減算でのボローが発生したときに、セットされ、その他のときにクリアされる。他のフラグは ABCD 命令と同じ変化をする。

次のプログラムは図 13.9 に示される 4 バイト 2 進化 10 進数の減算を行うルーチンである。

```

ADDQ .L    #4, A0          下位バイト ポイント
ADDQ .L    #4, A1          下位バイト ポイント
MOVEQ .L   #3, D0         ループ カウンタ セット
MOVE      #$04, CCR        CCR, Z フラグ セット
L01:  SBCD  -(A0), -(A1)    10 進減算
      DBRA  D0, L01

```

このルーチンを出たあと、Z フラグは結果の 4 バイト 2 進化 10 進数がゼロであるかどうかを示している。C フラグは結果が真数であるか、負の数の補数表示であるかを示して



図 13.10 2進化10進数減算の数値例

† SBCD  $D_y, D_x \rightarrow D_x - D_y \rightarrow D_x$   
 SBCD  $-(A_y), -(A_x) \rightarrow (A_x) - (A_y) \rightarrow (A_x)$

いる。図 13.10 にこのような減算の数値例を示す。この補数表示を真数表示に直すには次の NBCD 命令を使えばよい。

c. NBCD (Negate Digit) 命令<sup>t1</sup> この命令は 2 進化 10 進数の 10 の補数または 9 の補数をとる命令である。この命令では、デスティネーションオペランドの内容と X フラグをゼロから引いて、デスティネーションに格納する。したがって X フラグがクリアされているときには 10 の補数が得られ、また X フラグが立っているときには 9 の補数が得られる。オペレーションサイズはバイト長だけが許されている。デスティネーションオペランドのアドレス形式は、アドレスレジスタとディスプレースメント付/インデックス付プログラムカウンタ相対形式を除いて他のすべてが可能である。コンディションコードは SBCD 命令と同様の変化をする。

### 13.4 テストアンドセット命令

TAS (Test and Set Operand) 命令<sup>t2</sup> テストアンドセット命令は、オペランドの値をテストし、同じバスサイクルでオペランドのビットをセットする命令である。この命令の動作を図 13.11 に示す。この命令はマルチプロセッサシステムで共通メモリを誤りなく参照するために便利である。

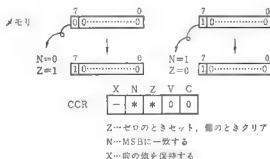


図 13.11 TAS 命令の動作

オペランドサイズはバイトのみであり、テストの結果はコンディションコードの N フラグまたは Z フラグに反映される。

次にマルチプロセッサシステムで共通メモリを参照する方法を説明する。図 13.12 に示

<sup>t1</sup> NBCD <ea> 0-<ea> → <ea>

<sup>t2</sup> TAS <ea> <ea> テスト → cc; <ea> の第 7 ビットに 1 をセット

のように2つのプロセッサ MPU-A と MPU-B が共通メモリをもっており、このメモリ中に2つのプロセッサが共有する変数があるとす。共有変数として時刻を表す変数 hh, mm, ss を考えよう。MPU-A は外部からクロックの割込みにより時刻変数を更新し、また MPU-B は必要に応じて時刻変数を参照する。

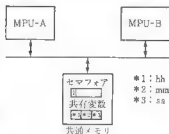


図 13.12 マルチプロセッサシステム

いま時刻が 10 時 59 分 59 秒とすると時刻変数の値は hh=10, mm=59, ss=59 となっている。外部からの割込みがあると MPU-A は時刻変数

の値を更新しはじめる。ss=00, mm=00 まで更新したときに、たまたま MPU-B が時刻変数を参照すると、10 時 00 分 00 秒と読み取られ、本当の時刻 11 時 00 分 00 秒とは異なり、誤りとなる。

誤りを防止するには共有変数を同時に2つのプロセッサからアクセスしないようにすればよい。これを相互排除 (Mutual Exclusion) の原理という。

ソフトウェアで相互排除を行うには、共有変数の参照状態を示すセマフォ<sup>†</sup>とよばれるフラグを用いる。セマフォの値が1であるときはそのセマフォに対応する共有変数がどれかのプロセスから参照されていることを示す。各プロセッサは共有変数を参照するときにはセマフォを調べ、1であれば他のプロセッサの参照が終るまでまつ。0であればすぐに1を立てて他のプロセッサに共有変数が参照中であることを示す。参照を終了するときにはセマフォを0に戻さなければならない。セマフォを調べるとき TAS 命令の代わりにテスト命令とセット命令を用いると相互排除ができない。なぜなら一方のプロセッサのテスト命令とセット命令の実行の間に、他方のプロセッサが割り込んで同時に共有変数の参照に入る可能性があるからである。

以上から共有変数を参照するルーチンの例は次のように書ける。

L01:	TAS	SEM	セマフォの値を CCR に反映し、1 を立てる
	BMI.S	L01	1 であればループして待つ
			0 であれば共有変数の参照に入る
		}	このときセマフォは1になっている
	CLR	SEM	セマフォを0にして参照を終了する

<sup>†</sup> セマフォとは鉄道の扉式信号機を意味する言葉である。

# 14

## 桁移動, ビット操作命令

この章ではレジスタやメモリの内容を桁移動したり、桁送りしたりする命令およびレジスタやメモリの特定ビットを判定したり、操作を加えたりする命令について述べる。前者はシフト、ローテート命令グループ、後者はビット操作命令グループに分けられる。68000では4つのシフト命令 (ASL, ASR, LSL, LSR 命令)、4つのローテート命令 (ROL, ROR, ROXL, ROXR 命令) および4つのビット操作命令 (BTST, BSET, BCLR, BCHG 命令) が用意されている。これらの命令はバイト、ワード、ロングワードのオペランドサイズや豊富なアドレス形式を組み合わせることができ、多くの操作を容易に実現できる。

### 14.1 論理形桁移動 (論理シフト) 命令

**LSL, LSR (Logical Shift Left/Right) 命令†** レジスタやメモリの内容を桁移動させる操作が論理シフトである。68000には2つの論理シフト命令 **LSL, LSR** 命令が用意されている。これらはデスティネーションオペランドのバイト、ワードあるいはロングワードのデータを単純に左へ (LSL) または右へ (LSR) 動かすものである。このときシフトされてオペランドの外へ出てしまったビットは順次コンディションコードレジスタ内にある C フラグおよび X フラグに移り、もとのオペランドレジスタからは捨てられてしまふ。逆の端のビット位置には0が入る。68000の論理形桁移動命令はデータレジスタの内容をシフトする場合とメモリの内容をシフトする場合とがあり——算術形桁移動命令、循環形桁送り命令も同様——両者の機能が若干異なる。

---

† LSL/LSR <データ>, Dy Dy の内容を <データ> だけシフト  
LSL/LSR Dx, Dy Dy の内容を Dx の内容だけシフト  
LSL/LSR <ea> <ea> の内容を1ビットだけシフト



まず、データレジスタの内容のシフトについて述べる。この場合オペランドサイズはバイト、ワード、ロングワードが許されている。データレジスタの内容をシフトするためには、ソースオペランドで桁移動の数を指定し、デスティネーションオペランドで被シフトデータレジスタを指定する。ソースオペランドの指定、すなわち桁移動の数の指定は次の2とおりの方法による。

(1) イミディエイトデータ（スタティック） シフトカウントをイミディエイトデータで直接指定する。シフトカウントは1～8の範囲に限られる。

(2) データレジスタ（ダイナミック） シフトカウントをデータレジスタにセットし、そのデータレジスタを指定する。シフトカウントはそのデータレジスタの下位6ビットに格納された値が有効となり、上位ビットは無視される。したがってシフトカウントは0～63の範囲となる。たとえばデータレジスタに67が格納されていたとすると、これは2進数、0100011であるから下位6ビット分の値は000011、すなわち3がシフトカウントになる。

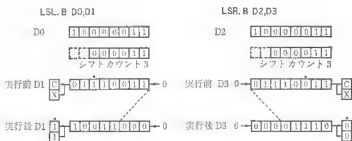


図 14.1 論理シフト命令 LSL, LSR の例

図 14.1 は左論理シフト命令 LSL および 右論理シフト命令 LSR でデータレジスタ D1 に格納された \$73 を左右論理シフトした例である。シフトカウントはデータレジスタ D0 で指定する。データレジスタには 16 進数 \$83 が格納されているが、6 ビット ( $2^6 - 1 = 63$ ) 分のデータのみが有効となりシフトカウントは 3 となる。LSL 命令によってオペランドの最上位ビットからシフトされて出てしまったビットは C フラグおよび X フラグに移され、LSR 命令では最下位ビットからシフトされて外へ出たビットが C フラグおよび X フラグに順次移されるので、命令実行後 LSL 命令の例では 1 が、LSR 命令の例では 0 が C フラグおよび X フラグに格納されている。LSL の場合の “1” は命令実行前のデータレジスタ D0 の第 6 ビット ( $2^6$ ) の “1” を反映し、LSR の場合の “0” は命令実行前の

データレジスタ D3 の第3ビット ( $2^3$ ) の“0”を反映したものである。C フラグおよび X フラグはこのように最後にシフトされて外に出たビットの値が格納される。ただし、シフトカウントが0のときは、C フラグは常に0となり、X フラグは変化せず前の値が保存されるようになっている。

シフト カウントが1~8の範囲においては、イミディエイト データによって直接シフト カウントを指定した方が便利である。データ レジスタにシフト カウントを設定する必要がなく、単一命令で桁移動を実行できるからプログラムのステップ数が小さくでき、処理時間も短縮できる。

次に、メモリの内容をシフトする場合について述べる。シフト カウントは常に1ビットであり、オペランドサイズもワード だけに限られる。したがってこの場合はシフト カウントやオペランド サイズを指定することはできない。これらを指定して論理シフトしたい場合はメモリ上のデータを一旦データ レジスタに取り出したうえで前述の論理シフトを行えばよい。

論理形桁移動は符号なし整数の乗・除算に用いることができる。また各種のデータ中の特定の部分を操作してデータを加工したり、並べ換えたりするのににも用いられる。次のルーチンは論理シフトを用いて乗算を行う例で、符号なしの16ビット(ワード)データを10倍するものである。乗算結果が符号なしの16ビットに収まるには被乗数は6553以下である必要がある。

MULT	MOVE. W	DATA 1, D0	被乗数取出し
	LSL. W	#1, D0	被乗数×2
	MOVE. W	D0, D1	退避
	LSL. W	#2, D0	被乗数×8
	ADD. W	D1, D0	被乗数×(2+8)=被乗数×10

LSL 命令により1ビット左へ論理シフトすることは2倍することに相当し、nビット左へ論理シフトすることは $2^n$ 倍することに相当する。また1ビット右へ論理シフトすることは1/2倍、すなわち2で割ることに相当し、nビット右へ論理シフトすることは $1/2^n$ 倍、すなわち $2^n$ で割ることに等価である。次の例は符号なしの16ビット(ワード)データを8で割るルーチンである。除算結果は小数点以下が切り捨てられる。

DIVD	MOVE. W	DATA 1, D0	被除数取出し
	LSR. W	#3, D0	被除数÷8

メモリ上の16ビット(ワード)データを直接8で割るには、1ビットの右論理シフトを3回繰り返せばよい。この場合はデータ レジスタを介さないで除算が実行される。

DIVM	LSR, W	DATA 1	被除数÷2
	LSR, W	DATA 1	被除数÷2
	LSR, W	DATA 1	被除数÷2

除算結果はもとのメモリ上の同一番地 DATA 1 に与えられるため、被除数は退避しておかないかぎり失われてしまう。

次のルーチンはメモリ上の8ビット（バイト）データを2桁の16進数表示で外部装置に出力するため、ASCII 符号に変換するものである。論理シフトデータの一部分が処理に用いられている。

BIAS	MOVE, B	DATA 1, D1	16進数の2桁の取出し（上位桁）
	LSR, B	#4, D1	下位4ビットを消す
	ORI, B	#\$30, D1	ASCIIに変換
	CMPI, B	#\$3A, D1	A~F 補正か？
	BLT, S	BS 1	補正不要なら分岐
	ADDI, B	#7, D1	A~F 補正
BS 1	MOVE, B	D1, BUF 1	変換結果の退避（上位桁）
	MOVE, B	DATA 1, D1	16進数2桁の取出し（下位桁）
	LSL, B	#4, D1	上位ビットを消す
	LSR, B	#4, D1	変換準備
	ORI, B	#\$30, D1	ASCIIに変換
	CMPI, B	#\$3A, D1	A~F 補正か
	BLT, S	BS 2	補正不要なら分岐
	ADDI, B	#7, D1	A~F 補正
BS 2	MOVE, B	D1, BUF 2	変換結果の退避（下位桁）

以上により DATA 1 に格納されていた8ビット（バイト）データ2桁の16進数表示の ASCII 符号 0~9, A~F に変換され、BUF 1 と BUF 2 に出力される。

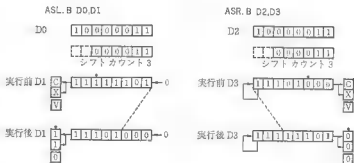
逆に16進数表示のデータの ASCII 符号が与えられたとき、これを2進データに変換するルーチンは論理シフトを用いて同様にプログラミングすることができる。これについては19章のプログラム例を参照されたい。

## 14.2 算術形桁移動（算術シフト）命令

ASL, ASR (Arithmetic Shift Left/Right) 命令† 算術形桁移動は2の補数表示さ

† ASL/ASR	#(データ), Dy	Dy の内容を #(データ) だけシフト
ASL/ASR	Dx, Dy	Dy の内容を Dx の内容だけシフト
ASL/ASR	<ea>	<ea> の内容を1ビットだけシフト

れた2進数を算術的、すなわち符号を保ったまま桁移動できることが特徴であり、この点が単純にオペランドを左右に桁移動する論理形桁移動と異なっている。68000では2つの算術シフト命令、すなわち左算術シフト **ASL** 命令と右算術シフト **ASR** 命令とが用意されている。ASL 命令は論理シフト **LSL** 命令と同様にオペランドを左へシフトするもので、左端の最上位ビットから外へ出されたビットは順次 **C** フラグおよび **X** フラグに移される。LSL 命令との違いは、シフトの際に符号の変化が生じた場合、コンディションコードの **V** フラグが1に設定されることである。これによりシフトデータの符号の変化を知ることができる。ASR 命令は論理シフト **LSR** 命令と同様にオペランドを右へシフトするもので、右端の最下位ビットから外へ出されたビットは順次 **C** フラグおよび **X** フラグに移される。このとき最上位ビットすなわち符号ビットは不変でありこれが順次右へシフトされていく。なお ASR 命令では符号ビットは不変であるから、**V** フラグは常に0である。ASL/ASR 命令は **LSL**/**LSR** 命令と同じくデータレジスタの内容をシフトする場合とメモリの内容をシフトする場合とがある。前者の場合、オペレーションサイズはバイト、ワード、ロングワードが許されており、シフトカウントの指定方法も **LSL**/**LSR** 命令と同じく (1) イミディエントデータ(スタティック形)で指定する方法と、(2) データレジスタ(ダイナミック形)で指定する方法とがある。また後者の場合は、オペレーションサイズがワードのみに限られ、シフトカウントも1ビットのみである。



■ 14.2 算術シフト命令 ASL, ASR の例

図 14.2 にデータレジスタでシフトカウントを指定し、符号ビット付の8ビットデータ、-3、-24 を左右算術シフトした例を示す。16ビットデータ、-3、-24 は16進数表示の \$FD, \$E8 にそれぞれに相当する。左算術シフト命令 **ASL** では \$FD、すなわち -3 を左へ3ビットシフトして -24 を得ている。これは -3 が8倍された結果である。V フラグは0で符号ビットの変化はなく、得られた値は算術シフトによる正しい結果を示してい

る。Vフラグが1になるような算術シフトにおいては、一般に正しい結果が得られていないので注意が必要である。一方右算術シフト命令 ASR では SEB すなわち -24 を右へ3ビットシフトとして -3 を得ている。これは -24 が 1/8 倍された結果である。ASR 命令では本来の算術シフトによる結果が常に得られると考えてよい。シフト実行前のオペランドデータ、すなわち被除数が負のときは符号ビットは1に保たれ、右シフトによって空いた上位ビットには1が埋められる。被乗数が正のときは符号ビットは0に保たれ、空いた上位ビットには0が埋められる。この結果 n ビット右算術シフトによって符号が変わることはなく、 $2^n$  で除算された結果がそのままレジスタに得られるわけである。ただし小数点以下はレジスタの外へ棄てられてしまう。正数の算術形右シフトでは除算の商をレジスタに残し、余りは棄てられる。たとえば  $1101_2$  を3ビット右シフトすると  $0001_2$  となり13を8で割った商は1ということになる。したがって端数である5は失われる。

負数の算術形右シフトでは様子が異なる。たとえば  $11110011_2(-13)$  を3ビット右シフトすると、 $11111110_2(-2)$  となり、-13を8で割った商は負数 -2 で、端数は +3 ということになる。

算術シフトを用いると2の補数表示されたデータの簡単な乗除算を、正負の符号に拘らず、同一の手順で実行できる。次のルーチンは2の補数表示による16ビット(ワード)データを2.5倍するものである。右算術シフトで与えられたデータを2を割ったものと、左算術シフトで2倍したものとを加算して結果を得ている。

ASMD	MOVE. W	DATA 1, D0	データ取出し
	ASL. W	#1, D0	データ×2
	MOVE. W	D0, D1	退避
	ASR. W	#2, D0	データ÷2
	ADD. W	D0, D1	2.5倍の実行
	MOVE. W	D1, DATA 2	結果の退避

左算術シフト命令 ASL で1ビットシフトしたデータレジスタ D0 を右算術シフト命令 ASR で2ビットシフトすると、元のデータを2で割った値が得られる。上の例で ASL 命令と ASR 命令を入れ替えることもできるが、ASR 命令をさきに行うと小数点以下が失われるので数値計算上好ましくない。

### 14.3 循環形桁送り（ローテート）命令

ROL, ROR (Rotate without X Left/Right)<sup>11</sup>, ROXL, ROXR (Rotate through X Left/Right)<sup>12</sup> 命令 循環形桁送りはオペランドの左端と右端、すなわち最上位ビットと最下位ビットを連結して環状にし、この中でオペランドの桁送りを行うものである。オペランドのデータはこの環の中を回転（ローテート）し、一巡すると元の位置に戻る。論理桁移動でレジスタの外に出されたビットが反対側の端から順次レジスタに入ってくると考えてもよい。68000では4つのローテート命令がある。ROL 命令と ROR 命令はオペランドを左右にそれぞれローテートする命令である。ROXL 命令と ROXR 命令はエクステンドビット付ローテートとよばれ、オペランドの最上位ビットと最下位ビットの間に1ビットの X フラグを介在させ、この拡張ビットを含んだ環の中でオペランドの左右桁送りをそれぞれ行うものである。

ローテート命令においても、論理シフト命令、算術シフト命令と同様にデータレジスタを扱う場合とメモリを扱う場合がある。データレジスタの内容をローテートする場合には、オペレーションサイズはバイト、ワード、ロングワードが許される。ローテートカウンタは論理シフト、算術シフトの場合と同じく (1) イミディエイトデータ（スタティック形）と (2) データレジスタ（ダイナミック形）により指定することができる。一方、メモリの内容をローテートする場合、オペランドサイズは16ビット（ワード）に限られ、ローテートカウンタも1ビットに限定される。メモリ上のデータを直接 n ビット分ローテートしたいときは、1ビットのローテート命令を n 回繰り返して用いればよい。

図 14.3 に4つのローテート命令 ROL, ROR, ROXL, ROXR によって循環形桁送りを行った例を示す。ROL (ROR) 命令において最上位ビット（最下位ビット）からローテートされて外へ出てしまったビットは C フラグに入るとともに、反対側の最下位ビット（最上位ビット）に戻る。したがって命令実行後 C フラグには一番最後にローテートにより外へ

11	ROL/ROR Dx, Dy	Dy の内容を Dx の内容だけローテート
	ROL/ROR #<データ>, Dy	Dy の内容を #<データ> だけローテート
	ROL/ROR <ea>	<ea> の内容を1ビットだけローテート
12	ROXL/ROXR Dx, Dy	Dy の内容を X フラグとともに Dx の内容だけローテート
	ROXL/ROXR #<データ>, Dy	Dy の内容を X フラグとともに #<データ> だけローテート
	ROXL/ROXR <ea>	<ea> の内容を X フラグとともに1ビットだけローテート

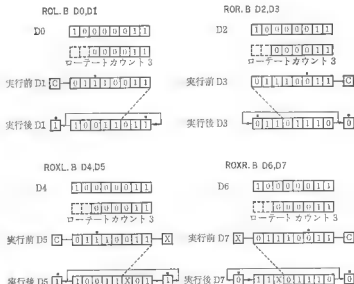


図 14.3 ローテート命令 ROL, ROR, ROXL, ROXR の例

出たビットの値が入っている。ただしローテート カウントが0のときは常に0となる。X フラグは命令実行前の値が保持される。ROXL (ROXR) 命令においては最上位ビット (最下位ビット) からローテートされて外へ出てしまったビットはC フラグとX フラグに入る。また反対端の最下位ビット (最上位ビット) にはX フラグの前の値がつけられる。命令実行後 C フラグ、X フラグには一番最後に外へ出されたビットの値が入っているが、ローテート カウントが0のときは、C フラグには命令実行前の X フラグの値が入り、X フラグの値は変化しない。

循環形桁送り命令によって16ビット データのパリティ ビットを求める方法を次に示す。

PRTY 1	MOVE.W	#15,D0	回数を設定
	CLR.B	D1	パリティ ビット クリア
PR 2	ROL	DATA1	1ビット ローテート
	BCC	PR1	1でなければ分岐
PR 1	EORI.B	#1,D1	パリティビット反転
	DBRA	D0,PR2	16回完了でなければ分岐

以上により DATA1 の "1" をとるビットの数が偶数ならばパリティ ビットは0となり、DATA1 の "1" をとるビットの数が奇数ならばパリティ ビットは1となる。この値

はデータレジスタ D1 の最下位ビットに得られる。循環形桁送りによって DATA1 のデータは破壊されないで残る。上のルーチンが終了したとき、DATA1 の値は最初の値に戻っている。

これまで述べてきた論理形、算術形シフト、およびローテートの各命令とその動作の概要は5章の表5.5にまとめている。

## 14.4 ビット操作命令

ビット操作命令はレジスタやメモリ内のデータの特定のビットの状態を調べたり、特定ビットの操作を行う命令である。68000 には BTST, BSET, BCLR, BCHG の4種の命令が用意されている。

**a. BTST (Bit Test) 命令<sup>11</sup>** BTST 命令はオペランドのビット群のうち、指定されたビットの状態を調べ、その結果を Z フラグに反映する。操作対象ビットの値が0であれば Z フラグは1となり、操作対象ビットが1であれば Z フラグは0となる。操作対象となるオペランドのサイズはデスティネーション オペランドに依存して決定され、ユーザが指定することはできない。すなわち、オペランドがデータレジスタの場合はロングワードとなり、指定されるビット番号は0~31が用いられる。オペランドがメモリ内のデータの場合はバイトサイズとなり、指定されるビット番号は0~7が用いられる。

ビット番号の指定には次の2通りの方法がある。

(i) イミディエイトデータ (スタティック) ビット番号をイミディエイトデータで直接指定する。オペランドがデータレジスタの場合は下位5ビットのデータが有効となる。すなわち (0~31) が用いられる。メモリ上のデータの場合は下位3ビットのデータが有効となる。すなわち (0~7) が用いられる。

(ii) データレジスタ (ダイナミック) ビット番号をあらかじめデータレジスタに設定しておき、そのデータレジスタを指定することにより操作対象となるビットを指定する。オペランドがデータレジスタの場合は下位5ビットのデータが有効となる。メモリ上のデータの場合は下位3ビットのデータが有効となる。

**b. BSET (Bit Test and Set) 命令<sup>12</sup>** BSET 命令はオペランドのビット群のうち指

- 
- |         |                   |                                  |
|---------|-------------------|----------------------------------|
| 11 BTST | Dn, <ea>          | <ea> のビットのうち Dn で示されるビットをテスト     |
|         | #<データ>, <ea>      | <ea> のビットのうち #<データ> 番目のビットをテスト   |
| 12 BSET | Dn, <ea>          | <ea> のビットのうち Dn で示されるビットを1にセット   |
|         | BSET #<データ>, <ea> | <ea> のビットのうち #<データ> 番目のビットを1にセット |



定された番号の1ビットの状態を調べ、その結果をZフラグに反映した後、この指定したビットを1にする命令である。すなわち指定したビットが1であれば0をZフラグに設定し、指定したビットが0であれば1をZフラグに設定する。このあとで指定したビットを1にする。オペランドサイズはBTST命令と同じくデータレジスタの場合32ビット（ワングワード）、主記憶内のデータでは1ビット（バイト）が自動的に決められる。ビット番号の指定も（1）イミディエイトデータ（スタティック形）と（2）データレジスタ（ダイナミック形）の2とおりがある。

c. BCLR (Bit Test and Clear) 命令<sup>t1</sup> BCLR命令はオペランドのビット群のうち指定された1ビットの状態を調べて、この結果をZフラグに反映したのち、この指定ビットを0にする命令である。オペランドサイズとビット番号の指定についてはBTST命令の場合と同様である。

d. BCHG (Bit Test and Change) 命令<sup>t2</sup> BCHG命令はオペランドのビット群のうち指定された1ビットの状態を調べて、これをZフラグに反映した後は、指定したビットを反転する命令である。すなわち指定したビットが0ならばこれを1に逆転し、1ならばこれを0に逆転するものである。この場合オペランドの他のビットは不変である。オペランドサイズとビット番号の指定についてはBTST命令の場合と同様である。4種のビット操作命令とその動作の概要はすでに表5.6にまとめてある。

16ビットデータのパリティビットを求めることは前述の循環形桁送り命令のほか、ビット操作命令によっても実現できる。次にそのルーチンを示す。この例では、データレジスタD0をループカウンタとビット位置指定用レジスタとして兼用している。

PRTY 2	MOVE.W	#15, D0	回数を設定
	CLR.B	D1	パリティ ビットクリア
	MOVE.W	DATA 1, D2	データ取出し
PR 2	BTST	D0, D2	Kビットをテスト (K=(D0))
	BEQ	PR 1	1でなければ分岐
	EORL.B	#1, P1	パリティ ビット反転
PR 1	DBRA	D0, PR 2	16回完了でなければ分岐

- |    |      |              |   |
|----|------|--------------|---|
| t1 | BCLR | Dn, <ea>     | <ea>のビットのうちDnで示されるビットをテストした後、0にリセット     |
|    | BCLR | #<データ>, <ea> | <ea>のビットのうち#<データ>番目のビットをテストした後、0にリセット   |
| t2 | BCHG | Dn, <ea>     | <ea>のビットのうちDnで示されるビットをテストした後、このビットを反転   |
|    | BCHG | #<データ>, <ea> | <ea>のビットのうち#<データ>番目のビットをテストした後、このビットを反転 |

以上により DATA1 の 16 ビットのうち “1” の値をとるビットの数が偶数ならばパリティビットは 0 となり, “1” の値をとるビットの数が奇数ならばパリティビットは 1 となる。この値はデータレジスタ D1 に得られる。主記憶内の 8 ビット (バイト) のデータのパリティを求める場合は BTST 命令で直接各ビットの “0”, “1” をテストできるので、データをレジスタに取り出して各ビットをテストしなくてすむ。

# 15

## プログラム制御命令

この章ではプログラム制御命令、すなわち分岐命令、サブルーチン操作命令および条件セット命令について説明する。同時にサブルーチン操作におけるパラメータ授受の代表的な方法についても述べる。また、LINK/UNLK 命令は本来プログラム制御命令ではなくデータ転送命令に分類されるべきものであるが、サブルーチン操作と密接なかわりあいがあるため本章で説明する。

### 15.1 分岐命令

68000 の分岐命令には JMP, Bcc, BRA, DBcc 命令の 4 種類がある。この中で JMP, BRA 命令は無条件分岐命令であり、Bcc, DBcc 命令は条件付ブランチ命令である。以下、各命令について説明する。

a. JMP (Jump) 命令† JMP 命令は無条件ジャンプ命令であり、分岐先アドレスの指定方法は 6 種類のアドレス形式を用いることができる。すなわちアドレスレジスタ間接形式、ディスプレースメント付アドレスレジスタ間接形式、インデックス付アドレスレジスタ間接形式、長または短絶対アドレス形式、ディスプレースメント付/インデックス付プログラムカウンタ相対形式を使用できる。

また、JMP 命令はディスプレースメント付プログラムカウンタ相対形式を除いて、アドレス空間全体に分岐することが可能である。ディスプレースメント付プログラムカウンタ相対形式は 16 ビットの符号付整数で表せる範囲 ( $PC-2^{15} \sim PC+2^{15}-1$ ) に分岐することが可能である。

---

† JMP <ea>    アドレスレジスタ ← PC

**b. Bcc (Branch Conditionally)<sup>11</sup>, BRA (Branch Always)<sup>12</sup> 命令** Bcc 命令は条件付ブランチ命令であり、条件を“cc”で表す。Bcc 命令で使用できる条件は常に真 (T)、常に偽 (F) を除く 14 種類である (5 章、表 5.7 参照)。たとえば、「等しい場合に分岐する」という条件を示す命令は BEQ (Branch Equal) と書く。

BRA 命令は無条件ブランチ命令であり、指定した番地へ無条件に分岐する。

Bcc, BRA 命令はディスプレースメント付/インデックス付プログラムカウンタ相対形式のみ使用できる。アセンブラで記述する場合は分岐先をラベルで指定する。アセンブラは分岐命令の相対距離を計算し、その結果をディスプレースメントとする。ディスプレースメントは 8 ビットと 16 ビットの 2 とおりがある。これは分岐する範囲によって決る。すなわち、分岐命令とラベルの相対距離が  $-2^7 \sim 2^7 - 1$  バイト以内の場合はディスプレースメントは 8 ビットでよく、分岐命令とラベルの相対距離が  $-2^{15} \sim 2^{15} - 1$  バイト以内の場合にはディスプレースメントは 16 ビットになる。Bcc, BRA 命令の命令長はディスプレースメントが 8 ビットの場合に 1 ワード命令になり、16 ビットの場合に 2 ワード命令になる。これをアセンブラ レベルで指定する場合には、コメントの後に“S” (ショートの意味) を付けることで区別する。すなわち、

BEQ.S <ラベル>

は 1 ワードのオブジェクトコードを生成し、

BEQ <ラベル>

は 2 ワードのオブジェクトコードを生成する。

次に Bcc 命令の使用例を示す。これは CMP 命令でデータレジスタ D0 と D1 を比較してコンディションコードに結果を設定し、Bcc 命令で条件をテストするものである。

CMP D0, D1

BEQ <ラベル>

また、Bcc 命令は ADD, SUB, MULS, DIVS 命令などの直後に置き、各演算の結果により処理を選択するときにも用いられる。

JMP, Bcc, BRA 命令の比較表を表 15.1 に示す。この表より、JMP 命令はアドレス空間全体に分岐できるので、JMP 命令と飛び先との相対距離が大きい場合に利用できる。逆に BRA 命令は相対距離が小さい場合に利用できる。さらに、Bcc 命令はテストコンディションを用いた条件分岐に利用できる。

<sup>11</sup> Bcc <ラベル> cc が真ならば <ラベル> へ分岐、偽ならば次の命令を実行

<sup>12</sup> BRA <ラベル> 無条件に <ラベル> へ分岐

表 15.1 JMP, Bcc, BRA 命令比較表

命 令		JMP 命令	Bcc 命令	BRA 命令
分岐する範囲	PC-128~PC+127バイト	1~2 ワード	1 ワード	1 ワード
	PC-32 768~PC+32 767	1~2 ワード	2 ワード	2 ワード
	アドレス空間全体	1~3 ワード	×	×
条 件 分 岐		×	○	×

e. DBcc (Decrement Counter and Branch until Condition True or Count=-1)

命令† DBcc 命令は反復処理を1命令で実現できる。DBcc 命令の実行は反復回数を数えるデータレジスタを用いて行う。分岐条件(5章, 表 5.7 参照)は16種類すべてが使用でき、分岐先のアドレス形式は、Bcc 命令と同様にディスプレースメント付/インデックス付プログラムカウンタ相対形式のみである。

DBcc 命令の実行手順を図 15.1 に示す。まず、指定された条件をテストする。その結果が真なら次の命令を実行する。偽ならデータレジスタから1を引く。データレジスタの値が-1なら次の命令を実行し、-1でなければラベルで示す番地へ分岐する。DBcc 命令ではデータレジスタのサイズをワードとして使用する。

Bcc 命令が真の場合に分岐するのに対し、DBcc 命令は偽の場合に分岐することに注意しておく必要がある。DBcc 命令は条件が常に偽 (F) の場合に、ニモニツが DBRA (Decrement and Branch Always) と書くことができる。すなわち、DBRA 命令は常にデータレジスタから1を引き、-1なら次の命令に進み、-1以外なら分岐を実行する命令である。

DBRA 命令と BRA 命令を使用した場合の反復処理について比較する。DBRA 命令を使用すると、

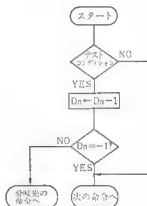


図 15.1 DBcc 命令の実行手順

† DBcc Dn, <ea> cc が真ならば次の命令へ進む。  
cc が偽ならば Dn ← Dn - 1 Dn = -1 ならば <ea> へ分岐。Dn = -1  
ならば次の命令へ進む。

```

      MOVEQ    #10, D0
LOOP   :
      DBRA     D0, LOOP

```

となり、1命令で反復処理を実現できる。これに対して **BRA** 命令を使用すると、

```

      MOVEQ    #10, D0
LOOP   :
      SUBQ     #1, D0
      BGE      LOOP

```

となり、**SUBQ**、**BGE** 命令の2命令かかる。反復処理ループ内から命令を1つ減じること  
は実行時間を短縮する上で効果が大きい。

また、**DBcc** 命令は反復処理のみに使用されるので **Bcc** 命令よりもプログラムの構造が  
単純化され、プログラムが見やすくなる。ただし **DBcc** 命令ではループカウンタ用のレ  
ジスタを専用に使用する。

## 15.2 サブルーチン操作命令

68000ではサブルーチン呼出し用の命令として **BSR**、**JSR** 命令の2つが用意されてお  
り、サブルーチンから戻る命令として **RTR**、**RTS** 命令が用意されている。以下、各命令  
について説明する。

**a. JSR (Jump to Subroutine) 命令<sup>t1</sup>** **JSR** 命令はプログラムカウンタの内容<sup>t2</sup>を  
システムスタックに格納し、その後指定されたアドレスへ分岐する命令である。**JSR** 命令  
の分岐先指定に使用できるアドレス形式は、**JMP** 命令と同様に、アドレスレジスタ間接形  
式、ディプレメント付アドレスレジスタ間接形式、インデックス付アドレスレジスタ  
間接形式、長および短絶対アドレス形式、ディスプレースメント付/インデックス付プロ  
グラムカウンタ相対形式の6種類である。

**JSR** 命令では、ディスプレースメント付プログラムカウンタ相対形式のときは、2ワ  
ード命令となる。

**b. BSR (Branch to Subroutine) 命令<sup>t3</sup>** **BSR** 命令はプログラムカウンタの内容

---

<sup>t1</sup> **JSR** <ea>      PC → SP @  
                                デスティネーション → PC

<sup>t2</sup> **JSR** 命令の次の命令を指している。

<sup>t3</sup> **BSR** <ラベル>    PC → SP @-  
                                PC+d → PC

をシステムスタックに格納し、その後指定したラベルに分岐する命令である。BSR 命令の分岐先指定に使用できるアドレス形式は、Bcc, BRA 命令と同様に、ディスプレイメント付/インデックス付プログラムカウンタ相対形式である。

また、BRA 命令は Bcc, BRA 命令と同様に分岐する範囲により命令長の選択ができる。すなわち、分岐する範囲が  $PC-2^7 \sim PC+2^7-1$  の場合には 1 ワード命令、 $PC-2^{15} \sim PC+2^{15}-1$  の場合には 2 ワード命令を選択できる。

**e. RTS (Return from Subroutine) 命令<sup>†1</sup>** RTS 命令はサブルーチンから戻るための命令である。RTS 命令を実行することにより、システムスタックから戻り先のプログラム実行番地が読み出され、プログラムカウンタにセットされる。

RTS 命令を用いるときのサブルーチンの動作例を図 15.2 (a) に示す。

一般に、この命令は、JSR, BSR 命令によってシステムスタックに退避されたプログラムカウンタの内容を回復する場合に用いられる。

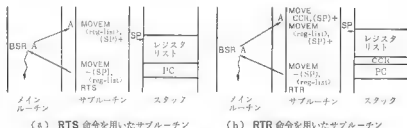


図 15.2 サブルーチン呼出し時の動作例

**d. RTR (Return from Subroutine and Restore CC) 命令<sup>†2</sup>** RTR 命令はシステムスタックからコンディションコードレジスタとプログラムカウンタの内容を回復する命令である。このときステータスレジスタのシステムバイト（上位 8 ビット）は影響を受けない。

RTR 命令を用いたサブルーチンの動作例を図 15.2 (b) に示す。この場合はサブルーチンの先頭で MOVE from SR 命令を用いてコンディションコードレジスタを退避しておく必要がある。

<sup>†1</sup> RTS (オペランドなし)     $SP@+ \rightarrow PC$

<sup>†2</sup> RTR (オペランドなし)     $SP@+cc$   
                                   $SP@+PC$

### 15.3 引数の受渡し方法

本節ではサブルーチンの引数の受渡し方法の代表的なものについて述べる。引数の受渡し方法には値呼び、番地呼び、アドレスリストのポインタを渡す方法などがある。以下にこれらの概要について述べる。

#### 15.3.1 値 呼 び

値呼びはパラメータを実際の値で受渡しするものであり、一般にメインルーチンからサブルーチン的一方通行の方法である。以下に値呼びを実現する方法を示す。

**a. データレジスタを用いる方法** データレジスタを用いて値呼びを実現する方法は操作も簡単で、実行時間も短い。すなわち、68000のアドレス形式の中で最も実行時間の短いデータレジスタ直接アドレス形式を用いることにより、引数の受渡しにかかる時間を短縮できる。データレジスタを用いた値呼びのプログラム例を以下に示す。

```
MOVE.W  引数1, D0
MOVE.L  引数2, D1
BSR     〈サブルーチン〉
```

このプログラムを実行した場合のレジスタの状態を図15.3に示す。サブルーチンで引数を参照する場合はデータレジスタ直接アドレス形式を用いる。

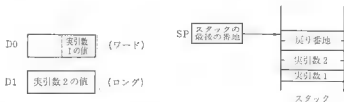


図 15.3 サブルーチンの引数（データ）をレジスタを用いて渡す方法

■ 15.4 サブルーチンの引数（データ）をスタックを用いて渡す方法

**b. スタックを用いる方法** 次にスタックを用いて値呼びを実現する方法を示す。メインルーチンで引数をスタックに格納する場合には、プリゲタリメントアドレスレジスタ間接アドレス形式を用いる。引数をスタックに格納した後にサブルーチンへジャンプする。スタックを用いた値引数のプログラム例を示す。

```
MOVE.W  引数1, -(SP)    引数1 → スタック
MOVE.L  引数2, -(SP)    引数2 → スタック
```



BSR      <サブルーチン名>

このプログラムを実行したときスタックの状態を図 15.4 に示す。スタックには MOVE 命令で引数が格納され、次に BSR 命令で戻り番地が格納される。サブルーチンで引数を参照する場合は、スタックポインタをベースレジスタとして用い、ディスプレースメント付アドレスレジスタ間接アドレス形式を用いる。すなわち、

```
MOVE.W 8(SP), D0    引数 1 → D0
MOVE.L 4(SP), D1    引数 2 → D1
```

で引数をデータレジスタに転送できる。

**e. プログラムの一部を用いる方法** プログラム領域中に仮引数の領域をとることにより値呼びを実現する方法を示す。この方法の利点はプログラム領域中に仮引数の領域をとるので、プログラムと仮引数の領域を別々に管理する手間が省けることである。通常仮引数の領域を JSR, BSR 命令の直後に置く。メインルーチンではサブルーチン呼び出す前に仮引数の領域に仮引数をセットしておく。たとえば、

```
BSR     <サブルーチン名>
DC.W    $10
```

とする。

サブルーチンからメインルーチンへ戻る場合には、図 15.5 に示すように JSR, BSR 命令の直後に戻るのではなく、仮引数の領域の次の番地へ戻らなければならない。そこで、JSR, BSR 命令でスタックに格納した戻り番地（プログラムカウンタの値）に仮引数の領域分だけ加える操作を行わなければならない。以下にプログラム例を示す。

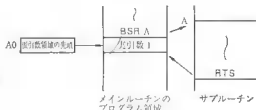


図 15.5 サブルーチンの引数(データ)をプログラムの一部を用いて渡す方法

```
MOVEA.L (SP), A0
ADDQ.L #2, (SP)
MOVE.W (A0), D1
```

まず、JSR, BSR 命令でスタックに格納した戻り番地はサブルーチン内で引数を参照するために必要なので、MOVEA 命令を用いてアドレスレジスタ A0 に格納する。次に、ス

タックに格納した戻り番地に仮引数の領域のバイト数を加える。引数を参照する場合は、A0 を用いアドレス レジスタ間接アドレス形式で参照する。

### 15.3.2 番 地 呼 び

番地呼びはパラメータをそれが格納されているメモリの番地で受け渡しするものであり、メインルーチンとサブルーチンの間の両方向のパラメータの受渡しが可能な方法である。以下番地呼びを実現する方法を示す。

**a. アドレスレジスタを用いる方法** 番地呼びでは実引数の番地が仮引数になるのでアドレスレジスタを用いる。アドレスレジスタに実引数の番地をセットするのに LEA 命令を用いる。以下にアドレスレジスタを用いた番地呼びの受渡し方法のプログラム例を示す。

```
LEA   引数 1, A0
LEA   引数 2, A1
BSR   〈サブルーチン名〉
```

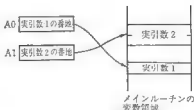


図 15.6 サブルーチンの引数（アドレス）をアドレスレジスタを用いて渡す方法

このプログラムを実行した場合のレジスタの状態を図 15.6 に示す。アドレスレジスタ A0, A1 には LEA 命令によって格納された実引数 1, 2 の番地が入っている。サブルーチンで引数を参照する場合はアドレスレジスタ間接アドレス形式を用いる。この方法はアドレスレジスタを引数の受渡しに使用できるようにレジスタの割当てを行

う必要がある。また、引数の数が多い場合にはよい方法ではない。

**b. スタックを用いる方法** 番地呼びをスタックを用いて実現するには PEA 命令を用いる。PEA 命令で実引数の番地をスタックに格納する。スタックを用いる方法のプログラム例を以下に示す。

```
PEA   引数 1
PEA   引数 2
BSR   〈サブルーチン名〉
```

図 15.7 にこのプログラムを実行したときのスタックの状態を示す。スタックには PEA 命令によって、実引数 1, 2 の番地が格納される。サブルーチン内でこれらの引数の番地を参照する場合には、スタックポインタをベースレジスタとして用い、ディスプレース

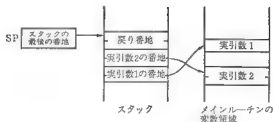


図 15.7 サブルーチンの引数  
(アドレス) をスタックを用い  
て渡す方法

メント付アドレス レジスタ間接アドレス形式を用いる。このプログラム例を以下に示す。

```
MOVEA.L 4(SP), A1
MOVEA.L 8(SP), A0
```

MOVEA 命令により、アドレス レジスタ A0, A1 に引数の番地がセットされる。実際に引数を参照するには、さらに、

```
MOVE.W (A0), D0
MOVE.L (A1), D1
```

とする。この MOVE 命令でデータ レジスタ D0, D1 に引数の値がセットされる。

### 15.3.3 アドレス レジスタをポインタとして渡す方法

この方法はテーブルに実引数のアドレスをあらかじめセットしておき、そのテーブルの先頭番地をアドレスレジスタにセットして、引数の受渡しを1個のアドレス レジスタで行うものである。この方法の利点は引数の受渡しにアドレス レジスタ1個で済むことである。アドレス リストをポインタで渡す場合のレジスタ、テーブルの状態を図 15.8 に示す。

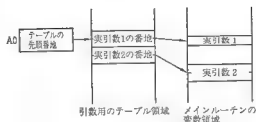


図 15.8 サブルーチンの引数をア  
ドレス リストのポインタを用いて  
渡す方法

たとえば、テーブルの先頭番地が1000 番地の場合は、

```
MOVEA.L #1000, A0
```

でアドレス レジスタ A0 にテーブルの先頭番地をセットできる。

次に引数を参照するプログラムを示す。



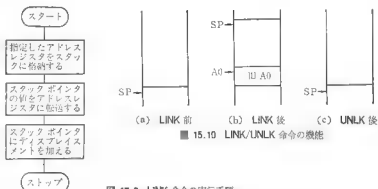


図 15.9 LINK 命令の実行手順

値が加えられる。すなわち、アドレスレジスタが確保された領域の先頭を指し、システムスタックポインタが最後を指すことになる。

LINK A0, # -10

という LINK 命令を実行した結果を図 15.10 (a), (b) に示す。アドレスレジスタ A0 が確保した領域の先頭を指しており、システムスタックポインタ SP が最後を指している。

**b. UNLK (Unlink) 命令** UNLK 命令は LINK 命令で確保したシステムスタック領域を開放するために用いる。UNLK 命令のデスティネーションオペランドにはアドレスレジスタを指定する。

UNLK 命令の実行手順を示す。オペランドで指定したアドレスレジスタの値 (LINK 命令を実行する前のシステムスタックポインタの値 -4) をシステムスタックポインタに移す。次に、このシステムスタックポインタの指すアドレス内容 (LINK 命令のオペランドで指定したアドレスレジスタの LINK 命令を実行する前の値) を UNLK 命令のオペランドで指定したアドレスレジスタに移す。この操作でシステムスタックポインタの値は 4 増加する。すなわち、UNLK 命令により、LINK 命令で指定したアドレスレジスタとシステムスタックポインタが LINK 命令を実行する前の状態に戻る。

UNLK A0

を実行した結果を図 15.10 (c) に示す。システムスタックが LINK 命令実行前と同じ状態に戻っていることがわかる。

次に実際に LINK, UNLK 命令を用いたサブルーチンの構成例を示す。

↑ UNLK An    An → SP  
              (SP) + → An

## メインルーチン

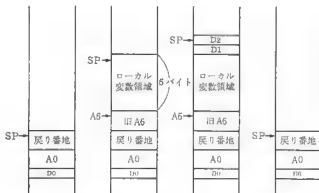
MOVE.W	D0, -(SP)	引数の受渡し (スタックを用いた値呼び)
PEA	(A0)	引数の受渡し (スタックを用いた番地呼び)
JSR	SUB	サブルーチン SUB へ分岐

## サブルーチン

SUB	LINK	A6, #-6	サブルーチン内で使用するローカル変数領域を 6 バイト確保
	MOVEM	D1/D2, -(SP)	サブルーチン内で使用するデータレジスタ D1, D2 を退避
	⋮		
	MOVEM	(SP)+, D1/D2	データレジスタ D1, D2 を回復
	UNLK	A6	ローカル変数領域の開放
	RTS		メインルーチンへ戻る

まず、引数の受渡しを示す。この例ではスタックを用いた値呼びと番地呼びを用いている。値呼びの場合は MOVE 命令でシステムスタックに引数の値を格納し、番地呼びの場合は PEA 命令でシステムスタックに引数のアドレスを格納する。次に JSR (または BSR) 命令でサブルーチンに分岐する。このときのシステムスタックの状態を図 15.11 (a) に示す。

サブルーチンではサブルーチン内で使用するローカル変数領域をシステムスタック内に確保する。この例では LINK 命令を用いてローカル変数領域を 6 バイト確保している。このときのシステムスタックの状態を図 15.11 (b) に示す。そして、サブルーチン内で使用するレジスタの内容を MOVEM 命令でシステムスタックに退避する。この例ではデー



(a) JSR 実行後 (b) LINK 実行後 (c) MOVEM 実行後 (d) UNLK 実行後

図 15.11 サブルーチン呼出しシーケンス中のシステムスタックの状態

レジスタ D1, D2 を退避している。このときのシステム スタック状態を図 15.11 (c) に示す。

サブルーチン内の処理が終了してサブルーチンから戻る場合には、MOVEM 命令で退避しておいたレジスタの値を戻す。次に、UNLK 命令によって LINK 命令で確保したローカル変数領域を開放する。このときのシステム スタックの状態を図 15.11 (d) に示す。

このように LINK, UNLK 命令を用いることにより、サブルーチン内で用いるローカル変数領域を動的に確保したり、開放したりすることが可能となる。

## 15.4 条件セット命令

**Scc (Set Conditionally) 命令†** Scc 命令は演算の結果をプログラムの別の場所で行いたい場合に使用する命令である。すなわち、Scc 命令はテストコンディションを行い、その結果をメモリ内の任意の番地に書き込む命令である。Bcc 命令では、条件を判定してすぐに次の処理を選択する場合に有効であるが、Scc 命令はその処理を後で行ったリ、結果を記憶しておく場合に有効である。

Scc 命令は“cc”で表す部分にニモニックで条件を指定し、その指定した条件が真 (T) なら実効アドレスで示されたバイトにオール 1 (11111111) をセットし、条件が偽 (F) ならそのバイトをオール 0 (00000000) にセットする。条件は 16 種類 (5 章、表 5.7 参照) 使用できる。たとえば、“等しい”かどうかという条件をチェックする場合には、SEQ というニモニックで命令を書く。

Scc 命令で利用できるアドレス形式はデータレジスタ直接形式、アドレスレジスタ間接形式、ポストインクリメントアドレスレジスタ間接形式、プリデクリメントアドレスレジスタ間接形式、ディスプレースメント付/インデックス付アドレスレジスタ間接形式、長または短絶対アドレス形式である。また、利用できるオペランドサイズはバイトだけである。

Scc 命令を使用したプログラム例を以下に示す。

```
CMP    #4, D1
SLE     1000
```

このプログラムは D1 の値を 4 と比較し、D1 の値が 4 より小さい場合に 1000 番地をオール 1 にセットするプログラムである。

---

† Scc <ea> cc が真のとき オール 1 → <ea>, cc が偽のとき オール 0 → <ea>

# 16

## システム制御命令

この章ではシステム制御命令として、トラップを発生させる命令、ユーザ状態においてステータスレジスタ/コンディションコードレジスタを操作する命令について説明する。特権命令は17章にて説明する。このほか、NOP命令および未実装命令についても述べる。

### 16.1 トラップ発生命令

トラップ発生命令はプログラムからトラップを発生するための命令である。トラップが発生するとプロセッサは例外処理を行い、スーパーバイザ状態でトラップに対する処理プログラムを実行する(8章参照)。トラップ発生命令は次のような場合に使用される。

- (1) ユーザプログラムが入出力動作をOSに依頼するとき
- (2) オーバフロー、0除算などの処理プログラムを起動するとき
- (3) ユーザプログラムを終了しOSへ制御を移すとき

68000には3種類のトラップ発生命令、すなわちTRAP、TRAPV、CHK命令が用意されている。以下、各命令ごとに説明を行う。

a. TRAP (Trap) 命令† TRAP命令は無条件にトラップを発生させる命令であり、プロセッサは例外処理を行う。TRAP命令のオペランドにはトラップベクタ番号を記述する。トラップベクタ番号には0から15までの16とおりを用いることができる。これらの番号は小さい方から順に32~47番の例外ベクタに対応する。たとえば、次の命令の実行例を図16.1に示す。

TRAP #1

プロセッサはプログラムカウンタおよびステータスレジスタをスーパーバイザシステム

---

† TRAP #<トラップベクタ番号> 無条件にトラップを発生



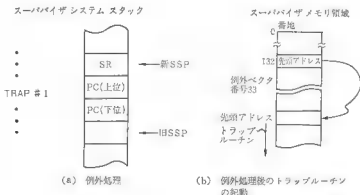


図 16.1 TRAP 命令の実行例

スタックに格納したのち、33番の例外ベクタ、すなわち132番地の内容で示される番地からトラップ処理を開始する（8章参照）。

トラップ処理ルーチンへパラメータを渡すためには、次のプログラム例のように TRAP 命令の直後にパラメータを設定する（15章参照）。この例では50がパラメータである。

```
TRAP    #10
DC, W   50
```

パラメータの受渡し方法は基本的にサブルーチン操作（15章参照）と同じである。

**b. TRAPV (Trap if Overflow Set) 命令<sup>†1</sup>** TRAPV 命令は演算のオーバーフローをチェックするための命令であり、Vフラグが1のときトラップが発生する。この命令にはオペランドはなく、例外ベクタ番号として7番（28番地に対応）が割り当てられている。オーバーフローの処理は28番地の内容で示される番地から実行される。オーバーフローが生じていなかった場合には次の命令の実行に移る。

**c. CHK (Check Register against Bounds) 命令<sup>†2</sup>** CHK はメモリアクセスの境界チェックを行うための命令である。境界の侵犯があった場合トラップが発生する。命令のフォーマットを次に示す。

```
CHK    <ea>, Dn
```

†1 TRAPV (オペランドなし)

†2 CHK <ea>, Dn    Dn < 0  
                    0 ≤ Dn ≤ <<ea>>  
                    Dn > <<ea>>

Vフラグ=1のときトラップ発生

N=1, トラップ発生

N=未定義, 次の命令へ進出

N=0 トラップ発生

ソース オペランドにはアドレス レジスタ直接形式以外のアドレス形式が使用できるが、デスティネーション オペランドはデータ レジスタを指定しなければならない。オペランドサイズはワードのみである。

この命令の実行は次のように行われる。プロセッサはデスティネーション オペランドで指定されたデータ レジスタ  $D_n$  を、0 ならびにソース オペランドで指定された  $\langle ea \rangle$  の内容と比較する。この比較結果により、命令の動作は次の3つの場合に分けることができる。

- (1)  $D_n < 0$  のとき、N フラグに1を設定しトラップを発生
- (2)  $0 \leq D_n \leq \langle ea \rangle$  のとき、N フラグは未定義。トラップは発生しない
- (3)  $D_n > \langle ea \rangle$  のとき、N フラグに0を設定しトラップを発生

トラップが発生した場合、プロセッサは例外処理を行い、例外ベクタの6番が内部的に生成される。この後、24番地で示される番地からCHKトラップ処理ルーチンが実行される。上記(1)、(3)のどちらの原因でトラップが発生したかはNフラグで判断できる。上記(2)の場合、プロセッサはCHK命令の次の命令を実行する。Z、V、Cフラグは(1)、(2)、(3)いずれの場合も未定義である。

CHK命令の使用例を次に示す。配列ARは\$300個のバイトデータの配列とし、アドレスレジスタA1はARの先頭番地を示しているとする。AR内のi番目 ( $0 \leq i \leq \$2FF$ ) のデータ  $AR(i)$  へアクセスするためには、iをデータレジスタ  $D_n$  にセットし、0(A1,  $D_n$ ) というインデックス付アドレスレジスタ間接形式を使用すればよい。このとき、データレジスタ  $D_n$  は、 $0 \leq D_n \leq \$2FF$  でなければならない。これをチェックするために

CHK    # \$2FF,  $D_n$

という命令が使用でき、配列外参照のチェックに役立つ。

## 16.2 コンディションコード操作命令

ユーザ状態でコンディションコードにアクセスする命令として68000には次の命令がある。

- (1) CCRと実効アドレス(EA)間の転送——MOVE EA to CCR, MOVE SR to EA
- (2) CCRに対する論理演算——ANDI to CCR, EORI to CCR, ORI to CCR

以下、各命令について説明する。

- a. MOVE EA to CCR (Move EA to Condition Codes Register) 命令<sup>71</sup>    この

命令は、実効アドレス EA の内容をコンディションコードレジスタへ転送する命令である。ソースオペランドのサイズはワードであるが、下位 8 ビットのみが CCR へ転送される。実効アドレス EA のアドレス形式はアドレスレジスタ直接形式が禁止されていることを除けば MOVE 命令と同じである。この命令は、CCR の各フラグに値を設定するときを使い、たとえば、10 進演算を行う前に、Z フラグを 1 に設定し、他のフラグをクリアしたい場合

MOVE #4, CCR

とすればよい。

b. MOVE SR to EA (Move from Status Register to EA) 命令<sup>12</sup> この命令はステータスレジスタの内容を実効アドレス EA へ転送する命令である。オペランドサイズはワードのみが許されている。実効アドレス EA のアドレス形式は通常の MOVE 命令と同じである。この命令は CCR をメモリ上に記憶するために使用する。たとえば、CCR をシステムスタック内へ格納する場合

MOVE SR, -(SP)

という命令を実行すればよい。この命令例はサブルーチンの先頭で使用される。サブルーチンから戻るときに RTR 命令を用いれば、コンディションコードレジスタもサブルーチン呼び出し前の状態に復帰できる。

c. ANDI to CCR (AND Immediate to Condition Codes) 命令<sup>13</sup> この命令は、コンディションコードレジスタの内容とイミディエイト値の論理積を計算し、コンディションコードレジスタへ格納する命令である。オペランドサイズはバイトのみである。たとえば、X フラグのみをクリアし、他のフラグは元の値のままにするためには、次の命令を実行すればよい。

ANDI #15, CCR

また、本命令の実行時間は 8 サイクルであり、MOVE EA to CCR 命令の実行時間は (12+EA 計算) サイクルである (7 章参照)。したがって、特定のフラグをクリアするためには ANDI to CCR 命令の方が MOVE EA to CCR 命令よりも 4 クロック + EA 計算の分だけ短い。

---

t1	MOVE <ea>, CCR	(<ea>) → CCR
t2	MOVE SR, <ea>	SR → <ea>
t3	ANDI #<データ>, CCR	CCR・#<データ> → CCR

#### d. EORI to CCR (Exclusive OR Immediate to Condition Code Register) 命令<sup>11</sup>

この命令はコンディションコードレジスタの内容とイミディエイト値の排他的論理和を計算しコンディションコードレジスタへ格納する命令である。オペランドサイズはバイトのみである。たとえば、Xフラグのみを反転させたいときは次の命令を実行すればよい。

EORI #16,CCR

e. ORI to CCR (OR Immediate to Condition Code Register) 命令<sup>12</sup> この命令はコンディションコードレジスタの内容とイミディエイト値の論理和を計算し、コンディションコードレジスタへ格納する命令である。オペランドサイズはバイトのみである。たとえば他のフラグに影響を与えずXフラグに1を設定したいときは次の命令を実行すればよい。

ORI #16,CCR

### 16.3 その他の命令

本節では、前節までに説明できなかった命令のうち、NOP命令ならびに未実装命令について説明する。

a. NOP (No Operation) 命令<sup>13</sup> NOP命令は何もしない命令である。命令長は1ワードであり実行時間は4クロックサイクルである。この命令は主として、プログラム中の領域確保ならびに、時間遅れとして利用される。

b. 未実装命令 未実装命令はオペレーションワードの上位4ビットが“1010”または“1111”のビットパターンをもつ命令である(8章参照)。8章でも説明したとおり、未実装命令は例外処理を引き起す。そのため未実装命令処理ルーチンに何らかの処理を行うプログラムを割り当てておけば、未実装命令が未実装処理ルーチン中のある種の処理を行うようにみえる。すなわち、いわゆるエミュレーションを行わせることが可能となる。

例として浮動小数点演算エミュレーション命令のフォーマットを図16.2に示す。図16.2の命令はデータレジスタD0とD4の浮動小数点演算を行うものであり、図中オペレーションフィールドで演算の種類を示す。第0~2および第5~8ビットの内容は0ま

<sup>11</sup> EORI #<データ>, CCR CCR ⊕ #<データ> → CCR

<sup>12</sup> ORI #<データ>, CCR CCR ∨ #<データ> → CCR

<sup>13</sup> NOP (オペランドなし) 何もしない

たは1のいずれであってもよい、オペレーションフィールドのビットパターンと演算の指定を次のように対応づける。

ビット4	ビット3		
0	0	FADD	浮動小数点加算
0	1	FSUB	浮動小数点減算
1	1	FMUL	浮動小数点乗算
1	1	FDIV	浮動小数点除算

データレジスタ D0 と D4 の浮動小数点加算命令をアセンブラ レベルで記述すると

**FADD D0,D4**

となる。この場合、アセンブラはこの命令を図16.2の命令フォーマットに変換できるものでなければならない。



図 16.2 浮動小数点演算のエミュレーション用命令フォーマット例

以下に、浮動小数点演算エミュレーション用の初期化ルーチンの例を示す。命令の上位4ビットが“1010”の命令は10番の例外ベクタを発生する。このため10番の例外ベクタに対応する40番地には初期化ルーチンの先頭アドレス1500番地を格納しておく。実際の初期化ルーチンは1500番地から始まる。この例では演算の種類を調べそれぞれの演算プログラムへジャンプさせるものである。なお、データレジスタD7およびアドレスレジスタA5の値は破壊される。

ORG	40	1010 命令の例外ベクトル番号
DC.L	1500	例外処理ルーチンの開始番地
* ORG 1500		
MOVEA.L	2(SP), A5	スタックから1010命令のプログラムカウンタA5へ転送
MOVE.W	(A5), D7	1010命令の命令コードをD7へ転送
MOVE.W	D7, -(SP)	1010命令の命令コードをスタックへ退避
ANDI	#\$18, D7	演算フィールド抽出
LSR	#\$1, D7	演算フィールドを0, 4, 8, 12の

			いずれかに変換
	LEA	OPANADDR, A5	オペレーションの先頭番地を A5 へロード
	MOVEA.L	0(A5, D7.W), A5	A5 の番地を演算フィールドで修 飾し、ジャンプ先を A5 へロード
	JMP	(A5)	各演算処理ルーチンへジャンプ
●			
OPANADDR	DC.L	FADD	加算処理ルーチン
	DC.L	FSUB	減算処理ルーチン
	DC.L	FMUL	乗算処理ルーチン
	DC.L	FDIV	除算処理ルーチン

## 17 特 権 命 令

8章では、68000に2種類のプログラム実行状態、すなわち、ユーザ状態とスーパーバイザ状態とがあることを説明した。本章では、スーパーバイザ状態（ステータスレジスタのSビットが“1”）でのみ使用可能な特権命令とよばれる命令について説明する。

68000でプログラムを実行状態を2種類に区別したのは、一般のプログラムの実行とシステムを管理するプログラム（一般にはOSと称されるシステムプログラム）の実行とを区別し、システム全体の信頼性を向上させるためである。この手法は、すでに大形計算機では一般的となっている。特権命令は、システムを管理するための命令であるので、特にこの命令の実行は他の命令の実行と区別する必要がある。特権命令を大きく分類すると、次のようになる。

- (1) ステータスレジスタの内容を変更 (ANDI, EORI, ORI, MOVE EA to SR 命令)
- (2) プログラム実行状態の変化に伴う相互間連絡 (MOVE to/from USP 命令)
- (3) タスクの切換えおよびその準備等 (RTE, RESET, STOP 命令)

特権命令はスーパーバイザ状態でのみ使用可能であり、ユーザ状態で使用しようとした場合には、命令実行の段階で特権例外が発生する。特権例外によって実行される特権違反例外処理に関しては8.3節(106ページ)を参照されたい。

a. ANDI to SR (AND Immediate to Status Register)<sup>†1</sup>, EORI to SR (Exclusive OR Immediate to Status Register)<sup>†2</sup>, ORI to SR (Inclusive OR Immediate to Status Register)<sup>†3</sup> 命令 これらの命令は、ステータスレジスタの内容 (S・Tビット、割込みマスク情報、コンディションコード) とイミディエイトデータとの論理演算を

†1 ANDI #〈データ〉, SR     $SR \cdot \# \langle \text{データ} \rangle \rightarrow SR$

†2 EORI #〈データ〉, SR     $SR \oplus \# \langle \text{データ} \rangle \rightarrow SR$

†3 ORI    #〈データ〉, SR     $SR + \# \langle \text{データ} \rangle \rightarrow SR$

行い、その結果に従ってステータスレジスタの内容を変更する命令である。この命令の実行により、プログラム実行状態の変更、トレース処理の実行、割込みレベルの設定、コンディションコードの初期設定が可能となる。本命令が、一般命令である“to CCR”と違う点は、“to CCR”の場合には無視されたイミディエイトデータ上位バイト（ビット番号15～8）が、本命令ではシステムバイトの情報を変更させる点である。システムバイトの情報は、プログラム実行状態の変更、トレース処理の実行、割込みレベルの設定を行うため、システムに直接影響を与える。

システムバイトの情報変更によるシステムコントロールの一例として、外部割込みによるタスク切換えの制御について簡単にふれる。たとえば、現在実行中のシステムプログラムおよびそれに続いてソフト的に起動されるプログラムにおいて、割込みレベル7以外の外部割込みは許したくないとする。この場合、システムバイトの割込みマスク情報だけを次の命令によって変更させ、レベル7とする。

**ORI    ## 0700, SR**

また逆に、すべての外部割込みに対して応答させようと考えた場合には、次の命令を実行して割込みマスク情報をレベルゼロに変更する。

**ANDI    ## \$ F 0FF, SR**

ファンクションコードFC0～2の出力をアドレスの一部として直接用いる場合には、ステータスレジスタSビットの情報を要する必要がある。すなわち、Sビットの情報がFC2の出力に直接反映されるため、スーパーバイザ状態で実行されるシステムプログラムがユーザ状態で実行されたプログラムによって生成されたメモリ上のデータを、参照できないという不都合が生じることがある。そこで、どうしてもやむを得ない場合この命令を用いてSビット情報を“0”に変更し、システムプログラムのプログラム実行状態をユーザ状態としてから、参照すべきデータを68000の内部レジスタ（D0～7）等に格納し、再度TRAP命令の実行によって実行状態をスーパーバイザ状態に遷移させる手続きが必要となる。しかし、68000上位16ビットマイクロコンピュータ68010では、スーパーバイザ状態においてはFC0～2の出力をプログラムで設定可能となり、この手続きも不要となる。

**b. MOVE EA to SR (Move EA to Status Register) 命令†** 本命令は、ステータスレジスタの内容を任意の値にセットする命令である。a. で説明した命令は、命令実行前のステータスレジスタの内容と論理演算するので、セットされる値は以前の状態を反映する。それに対し、この命令では、ステータスレジスタの内容を無条件に指定された値

---

† **MOVE <ea>, SR    <ea> → SR**



(ソースオペランド)とする。

たとえば、ステータスレジスタにオール0をセットしようとした場合を考える。ステータスレジスタをオール0にすると、プログラム実行状態はユーザ状態に、トレースは不実行の状態に、割込みマスクはレベル0に、そしてコンディションコードはすべて"0"の状態となる。これは、次の命令を実行すればよい。

MOVE    ##\$0000,SR

本命令では、ソースオペランドのアドレス形式として、アドレスレジスタ直接形式以外のすべての形式をとることができる。

**e. MOVE USP (Move User Stack Pointer) 命令<sup>1)</sup>**    本命令は、ユーザ状態とスーパーバイザ状態との間でシステムスタックポインタの値を連絡するためにある。68000ではシステムスタックポインタとしてアドレスレジスタA7を利用している。システムスタックポインタはユーザ状態ではUSP、スーパーバイザ状態ではSSPと区別されている。このように、アドレスレジスタA7は、実際には2つのレジスタで構成されており、どちらのレジスタを使用するかはプログラム実行状態(ステータスレジスタSビットの値)で自動的に決められてしまう。スーパーバイザ状態で実行するシステムプログラムにおいては、すべてこのデータを参照可能とする必要がある。そのため、スーパーバイザ状態で実行するプログラムで、ユーザ状態で使用したアドレスレジスタA7すなわちUSPをアクセスする手段が必要となる。この手段として利用されるのがこの命令である。

アドレスレジスタA0~6は、スーパーバイザ状態でも、ユーザ状態でも同じものである。したがってスーパーバイザ状態でのユーザ状態におけるアドレスレジスタA7 (USP)の参照は、これらアドレスレジスタA0~6を仲介として用いる。本命令は次に示すアドレス形式のみが許されている。

MOVE.L    An,USP    An→USP

MOVE.L    USP,An    USP→An

**d. RTE (Return from Exception) 命令<sup>1)</sup>**    本命令は、例外処理後に実行されるシステムプログラムから、例外処理を発生させたプログラムに制御を移すためのものである。8章の例外処理で説明したように、例外処理中に68000内のプログラムカウンタおよびステータスレジスタの内容はスーパーバイザシステムスタック上(SSPが指定するメ

<sup>1)</sup> MOVE An,USP    An→USP

MOVE USP,An    USP→An

<sup>2)</sup> RTE (オペランドなし)    (SP)+→SR

(SP)+→PC,ただしSPはSSP

メモリ上)に退避される。この退避された値を68000内に再び戻すと、例外処理を発生させた命令の次の命令を実行することができる。そのため、例外処理後に実行するシステムプログラムの最後で本命令を実行すれば、例外処理を発生させたプログラムの再開が可能である。

なお、アドレス エラー、バス エラーの例外処理においてシステム スタック上に退避されるデータは、プログラム カウンタおよびステータス レジスタの内容だけではないので、スタックポインタの内容はプログラム カウンタおよびステータス レジスタの格納アドレスを示していない。また、例外処理後のシステム プログラムでアドレス レジスタ A7 およびシステム スタック ポインタ（指定法は異なるが同一のレジスタ）を使用した場合にも同様の状況が考えられる。したがって、本命令を実行する前にはスタック ポインタの内容をチェックしておく必要がある。

**e. RESET (Reset External Devices) 命令<sup>11</sup>** 本命令を実行すると68000のリセットピン出力がアサートされ、このピンに接続された外部デバイスをリセットすることができる。リセットピン出力は124クロック分の期間アサートされている。外部デバイスのリセットは、本命令を実行し、124クロック分の期間に完了するようにする。

本命令の実行によって68000の内部レジスタが影響を受けることはない。また、次サイクル以降の命令読出しは、本命令の実行サイクルの一番最後で行われるため、リセット信号の供給が命令読出しに起因したバス エラーおよびアドレスエラーによって中断されることもない。本命令の実行では、リセット例外処理が行われない。

**f. STOP (Load SR Stop) 命令<sup>12</sup>** STOP 命令は、必要な割込みが発生するまでプロセッサをストップさせ、タスク間で同期をとるため等に使用される。

命令の実行では、まず、オペレーションワードに続く2バイトのイミディエイトデータをそのままステータス レジスタにセットする。これにより、ステータス レジスタ内の割込みマスクビット10~2（ビット番号8~10）もある値に設定される。その後、プロセッサは、この割込みマスクビットに設定したレベルより高レベルの外部割込み要求が行われるまで待機する（ストップ状態）。下記の命令を実行して、割込みマスク情報をレベル7にセットした場合には、レベル7の外部割込み要求および外部リセット信号の入力がないかぎり、プロセッサは再動作を行わない。

<sup>11</sup> RESET (オペランドなし) RES 端子をアサート

<sup>12</sup> STOP #〈データ〉 #〈データ〉→SR、割込みまでストップ状態

**STOP    ## 7F00    T="0", S="1", I0~I2="1", CC="0" を SR にセット、**  
**停止状態となる**

本命令によってタスク間で同期をとる場合としては次のような状況が考えられる。すなわち、68000 が所定の演算を終了し、I/O 待ちとなる場合や、複数の 68000 を用いてマルチプロセッサを構成し、処理の順序性が問題となるプログラムを実行する場合等である。このような状況の場合には、本命令を用い、割込み方式によって同期をとることが有効である。一方、このような機能をプログラムだけで実現しようとしたのが TAS 命令である。TAS 命令に関する詳細は 13 章を参照されたい。

# 18

## 例外処理後のシステム プログラム

8章で述べたように、68000では例外処理をプロセッサ内のマイクロプログラムにより実行する。しかし、その処理後にシステム運用上で必要となる種々の機能を果たすシステムプログラム(OS)は、それぞれのシステム構成に応じてユーザが用意しなければならない。このシステムプログラムの概要について、この章で簡単に説明する。

例外処理後に実行されるシステムプログラムのアドレスは、それぞれの例外処理発生要  
■別の例外ベクタによって指定される。例外処理では例外ベクタを読み出し、それぞれのシステムプログラムの実行準備までを行う(8章参照)。つまり、例外処理後に実行するシステムプログラムの起動を、例外処理によって行うのである。一方、例外処理後のシステムプログラムの実行によって、例外処理を発生させたエラー要因の回復を行う。さらにその後のタスクの変更(例外処理を発生させたプログラムの再起動あるいは全く別のプログラムの実行)もシステムプログラムが行うことになる。これは、RTE命令、RTR命令、RTS命令等を用いて実行する。

各例外処理後に実行されるべきシステムプログラムの基本機能、動作について以下説明していく。

### 18.1 リセット例外処理後のシステム プログラム

本システムプログラムの主要機能は大別すると、次の2点である。

- (1) システム全体の初期状態設定
- (2) 次に実行すべきプログラム(タスク)の選択

上記(1)の機能としては、イニシャルプログラムローディング(IPL)等があり、これらはシステム構成によって個々に準備する必要がある。一方、(2)の機能を有するシス

テム プログラムは一般にディスパッチャ (Dispatcher) とよばれている。

ディスパッチャは実行可能なプログラムのリストを調べ、その中で一番優先度が高いプログラムの実行を準備する。プログラム実行の準備として、システム スタック上に、そのプログラムの先頭アドレスおよび初期値とすべきステータス レジスタの内容等をセットする。その後、RTE 命令等を実行して、次に実行すべきプログラムに起動をかける。

実行可能なプログラムがない場合には、プロセッサを待機させる必要がある。実行可能なプログラムの発生を外部割込みの形でプロセッサに知らせる場合には、プロセッサに STOP 命令を実行させて割込み発生まで停止させればよい。また、実行可能なプログラムの発生をメモリ上のある特定なビットの変化によってプロセッサに知らせる場合には、無限ループ内で TAS 命令を実行して、そのビットをチェックし、その結果に従ってループを脱出するようにプログラムを構成すればよい。この無限ループ内のプログラムの実行が待機状態と同じ働きをする。

## 18.2 割込み例外処理後のシステム プログラム

本システム プログラムの主要機能は、外部デバイスとの通信である。たとえば、I/O 装置が入出力を終えた場合には、その終了を割込みによってプロセッサに知らせる。これら外部デバイスと 68000 との通信方式は、それぞれの外部デバイスに依存しているため一様ではない。そのため、それぞれの外部デバイスに適したシステム プログラムを起動する必要がある。

割込み例外処理によって起動されるシステム プログラムは、8.3 節で説明したように、割込みベクタを用いることにより複数個設定することができる。したがって、タイプの異なる外部デバイスごとに、それぞれに対応したシステム プログラムを準備することができる。そして、それらの起動は、ハード的 (あるいはソフト的) に割込みベクタを指定 (割込み例外処理中の割込みアクトレジサイクル) することで可能となる。

なお、割込み例外処理においてオートベクタを指定 (割込みアクトレジサイクルで  $\overline{\text{VPA}}$  をアサート) した場合には、ベクタ番号 25~31 が自動的に 68000 内で生成される。また、割込みベクタの読出しを失敗した場合には、スプリアス割込みにより、ベクタ番号 24 が自動的に生成される。これらの事項は、8.3 節で説明した。

### 18.3 トラップ例外処理後のシステム プログラム

本システム プログラムの主要機能は、ユーザ状態で実行されるプログラムからの I/O 要求、演算結果がオーバフローした場合の対策、メモリ間の管理等と種々雑多である。一般に、トラップ発生命令 (TRAP, TRAPV, CHK 命令) は、ユーザ状態で実行されるプログラムがスーパーバイザ状態で実行されるシステム プログラムを呼び出す (スーパーバイザコールと称す場合が多い) 場合に用いられることが多い。そのため、その機能はさまざまなものとなる。この種々雑多なシステム プログラムをそれぞれの目的に合せて呼び出せるよう、69000 では、トラップ例外処理内で次に行うシステム プログラムのアドレスを指定するために読み出す例外ベクタを 19 種 (TRAP 命令で 16 種、TRAPV, CHK 命令、除数 0 で各 1 種) 設定できるようにしている。

### 18.4 不当命令および未実装命令例外処理後のシステム プログラム

不当命令例外処理後のシステム プログラムの機能としては、次にあげる 2 種類等が考えられる。このうち、どの機能を選択するかはシステム設計者の判断で決る。

(1) 不当命令があったことをメモリ上に記録し、その命令の実行は行わず、次の命令から実行を再開する。

(2) 不当命令があったことをメモリ上に記録し、TRAP 命令によってディスパッチを起動させ、別のプログラム (タスク) の実行を行う。

一方、未実装命令例外処理後に起動されるシステム プログラムは、エミュレーション用のプログラムである。このことは、8.3 節、16.3 節でも説明した。この場合、このシステム プログラムではたとえば浮動小数点演算等をサポートし、最後に、RET 命令で例外処理を発生させたプログラムに戻ることになる。

### 18.5 特権違反例外処理後のシステム プログラム

本システム プログラムの主要機能は、前節で説明した不当命令例外処理後のシステム プログラムとはほぼ同様である。特権違反例外のあったことをメモリ上に記録し、その命令の実行を行わず次の命令の実行から再開、あるいは別のプログラム (タスク) を起動させる

ことがふつうである。

## 18.6 トレース例外処理後のシステム プログラム

本システム プログラムの主要機能は、1 命令実行ごとに 68000 内の情報をメモリ上に記録するトレース機能である。このトレース機能がプログラム開発の段階では大きな助けとなる（8.3 節参照）。トレース例外処理では、トレース機能を有する本システム プログラムを起動させる。

68000 内ユーザ レジスタの情報をメモリ上に格納するには MOVEM 命令が有効である。詳細に関しては 12 章を参照されたい。また、プログラム カウンタおよびステータス レジスタの内容はスーパーバイザ システム スタック（SSP がアドレスを指定するメモリ空間）上に格納されているので、これらの情報も併せて転送しておく、プログラムのデバッグにとって都合がよい。

## 18.7 バス エラー例外処理後のシステム プログラム

本システム プログラムの主要機能は、バス エラーからの回復である。バス エラー例外処理では、バス エラーを発生させた時点でのプログラム カウンタ、ステータス レジスタの内容の他に、そのとき実行していた命令のオペレーションワード、参照アドレスおよびそのアクセス タイプの情報がシステム スタック上に退避される（8.3 節参照）。したがって、これらの情報を有効に利用してエラー回復を図ればよいのであるが、現状 68000 では、プログラム カウンタによって示されるアドレスが、バス エラーを発生させた命令の拡張部から次に実行を予定している命令の拡張部までのどの部分を指定しているのか定義できない。そのため、一番簡単なエラー回復の手段はシステム全体をリセットするという方式である。68000 上位 16 ビットマイクコンピュータ 68010 のバス エラー例外処理の仕様は、もっとエラー回復に都合の良い仕様、すなわち、バス エラーを発生させた命令を再実行しやすい仕様に改善されることが予想される。

## 18.8 アドレス エラー例外処理後のシステム プログラム

本システム プログラムの主要機能は、アドレス エラーからの回復である。しかし、ア

ドレス エラー例外処理もバスエラー例外処理と同様、 退避されるプログラム カウンタの内容はエラー発生時のアドレスを限定できない。そのため、システム全体をリセットするという方式をとらずにエラー回復を図ろうとした場合には、オペレーション ワード、参照アドレスおよびそのアクセス タイプの情報を有効に利用する必要がある。

この章でのシステム プログラムの説明は概説的なものであり、これらプログラムの仕様はシステム構成、目的によって様々に変化する。そのため、この章で説明をしなかった機能がシステム プログラムに導入される場合もあれば、逆に、ここで説明した機能が省略される場合もある。現在、68000を用いたボードコンピュータ H680 SBC システムでは、前述のようなシステム プログラム(リアルタイムモニタ RMS, S680RMS 1R)を EPROM の形で提供している(Ⅲ編参照)。



# 19 プログラム例

前章までは主として各命令毎に機能の説明を行ったが、本章では各命令の理解を深めるために、いろいろな種類の命令を組み合わせた、より実際のプログラム例を示す。プログラム例として、2進化10進数の真数変換、1の個数の数え上げ、文字列の一致判定、ASCIIから2進数への変換、パリティの生成をあげる。これらはいずれも基本的なもののばかりであるが、前章までの補足という意味も兼ね備えている。各命令の説明を適宜参照することにより、理解が一層深まるであろう。

## 19.1 2進化10進数の真数変換

正の2進化10進数の減算において、被減数よりも減数のほうが大きいと演算結果は負になる。ところが、SBCD命令を用いてこの減算を行うと演算結果は10の補数表現になる。たとえば13章、図13.10で示したように12340000-12345678の減算ではボローが発生し、その結果は99994322である。本節では補数表現された演算結果を真数に直すプログラム例を示す。

補数を真数に直すには図19.1に示すようにゼロから補数を引けばよい。そこで、NBCD

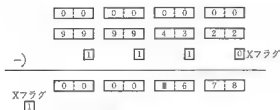


図 19.1 2進化10進数真数変換の数値例

命令を用いた次のルーチンを用いれば結果を真数に変換することができる。なお本来なら符号表示バイトを定義して、これを処理すべきであるがここでは省略した。なおレジスタ A1 は SBCD 命令の例と同様に 2 進化 10 進数 データの格納されたメモリを指し示すレジスタである。

	BCC	NEXT	減算結果で Borrow がなければ次へ
	ADDQ, L	#4, A1	下位桁を指し示す
	MOVEQ, L	#3, D1	ループカウンタをセット
	ANDI	#\$0F, CCR	X フラグをクリア
L01:	NBCD	-(A1)	10 進補数
	DBRA	D0, L01	ループ
NEXT:	EQU	*	補正終了

## 19.2 1 の個数の数え上げ

本節では、データレジスタ D0 中に含まれている 1 の個数を数え上げる方法について示す。D0 は 32 ビット長として使用する。したがって D0 に含まれる 1 の個数は 0 個から最大 32 個までである。D0 中の 1 の個数を数え上げるためには、論理シフトあるいはローテート命令を使う方法 (14 章参照) もあるが、本節では別の方法を示す。

図 19.2 はここで用いたアルゴリズムについて説明した図である。データレジスタ D2 には D0 から算術的に 1 を減算した内容が入っている。D0 では第 0 ビットから第 3 ビット

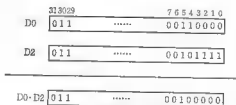


図 19.2 データレジスタ D0 に含まれる 1 の個数を数え上げる方法

ビットが 0 で第 4 ビットが 1 であるの

に対し、D2 では第 0 ビットから第 3 ビットまでが 1 で第 4 ビットが 0 であることに注目されたい。

したがって、D0 と D2 の論理積 D0-D2 においては、第 0 ビットから第 4 ビットまですべて 0 となる。これをもとの D0 と比較する

と、D0 に含まれる 1 のうち最も右にあるものが 1 個だけ 0 に反転していることがわかる。したがって、D0 と D0 から 1 だけ減算した D2 の論理積を新たな D0 とする操作を、D0 が 0 になるまで繰り返せば 1 の個数を数え上げることができる。D0 に含まれている 1 の個数が少ない場合には、このアルゴリズムは実行効率がよい。プログラム例を次に示

す。この例ではデータレジスタ D1 に 1 の偶数が入るものとする。

	CLR.L	D1	D1 をクリア
	TST.L	D0	D0 が 0 か?
	BEQ	QUIT	
L1:	ADDQ.L	#1, D1	D1 に 1 を加える
	MOVE.L	D0, D2	D0 へ D2 へコピーする
	SUBQ.L	#1, D2	D2 から 1 を減算する
	AND.L	D2, D0	D0 と D2 の論理積をとる
	BNE	L1	
QUIT:	NOP		

### 19.3 文字列の一致判定

本節では、長さ 80 文字 (80 バイト) の STRING1 および STRING2 の一致判定を行うプログラムを示す。各文字列はメモリ上にあるものとする。一致判定に際しては、CMPM 命令と DBNE 命令を用いる。プログラム例次に示す。

	LEA	STRING1, A0	A0 に文字列 1 の先頭番地をセットする
	LEA	STRING2, A1	A1 に文字列 2 の先頭番地をセットする
	MOVEI.L	#19, D0	D0 に文字列の長さ -1 をセットする
LOOP:	CMPM.L	(A0)+, (A1)+	文字列 1 と文字列 2 を 1 文字ずつ比較する
	DBNE	D0, LOOP	一致している間は分岐し比較を続ける
	BMI	D0, SAME	D0 はマイナスの値か?
	:		
STRING1:	DC	'COMPUTER...'	文字列 1 (80 バイト)
STRING2:	DC	'OVERFLOW...'	文字列 2 (80 バイト)

まず、LEA 命令を用いて 2 つの文字列の先頭番地をそれぞれアドレスレジスタ A0、A1 にセットする。次にデータレジスタ D0 に比較する文字列の長さ -1 (ここでは #19) をセットする。文字列の比較には CMPM 命令を用いる。CMPM 命令のアドレス形式はポストインクリメントアドレスレジスタ間接形式のみであり、命令実行毎にアドレスレジスタをインクリメントする。文字列が等しいかどうかは、DBNE 命令の実行結果でわかる。DBNE 命令で反復処理が終了する条件は文字列が一致しなかった場合もしくは設定された比較回数が終了した場合である。文字列が一致しなかった場合には、データレジスタ D0 の内容は -1 よりも大きくなる。設定された比較回数を終了した場合は文字列が等しかったことを示す。この場合にはデータレジスタ D0 の内容は -1 で終了する。

## 19.4 ASCII から2進数への変換

本節では16進数を表すASCII(0~9, A~F)4文字を16ビットの2進数へ変換するサブルーチンプログラムを示す。変換すべきASCII文字列は、図19.3に示すようにデ

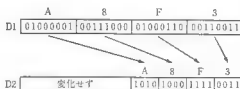


図 19.3 ASCII から2進数への変換

ータレジスタD1に上位桁から順に8ビットずつセットされているものとする。変換結果はデータレジスタD2の下位ワードにセットされる。

変換のアルゴリズムを次に示

す。まず\$4Fで第6ビットと下位4ビットの抜き出しを行う。ASCII文字が“A”~“F”か“0”~“9”かは第6ビット(上位から2ビット目)で判定できる。すなわち、第6ビットが1のとき“A”~“F”であり、0のとき“0”~“9”である。さらに、“A”~“F”の場合、抜き出したパターンから\$37を算術的に減算したものが\$A~\$Fになり、“0”~“9”の場合、下位4ビットがそのまま数値を表す。変換の順序は上位文字から始め、結果を格納するD2を4ビットずつが左シフトしていく。以下にプログラム例を示す。この例では、データレジスタがバイト、ワード、ロングワードのいずれでも扱えることを利用して、D1をデータの保持と演算の両方に用いている。なお、D1の内容は、変換終了後破壊されている。

ASCHEX:	CLR.W	D2	D2をクリア
	BSR.S	AHCV	最上位文字を変換
	BSR.S	AHCV	2番目の文字を変換
	BSR.S	AHCV	3番目の文字を変換
	BSR.S	AHCV	最下位の文字を変換
	RTS		結果がD2に入って終了
*			
AHCV:	ROL.L	#8,D1	D1の最上位バイトを最下位に移動
	ANDI.B	#\$4F,D1	下位4ビットと第6ビットを抜き出し
	CMPI.B	#\$10,D1	第6ビットが1か?
	BLT.S	AH1	下位4ビットが0~9ならAH1へ
	SUBI.B	#\$37,D1	A~Fなら10~15に変換
AH1:	LSL.W	#4,D2	D2を左に4ビットシフト
	ADD.B	D1,D2	D2シフトに結果の4ビットを加える

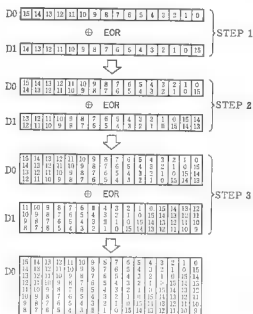
## RTS

## サブルーチン終了

## 19.5 パリティの生成

本節ではビットデータのパリティを生成するプログラム例を示す。ビットデータのパリティは、それぞれのビット値の排他的論理和により生成する。すなわち、パリティは、ビットデータ内の“1”の総数が偶数個の場合は0に、奇数個の場合は1になる。

ここでは、データレジスタD0の下位16ビットにビットデータが格納されているものとし、データレジスタD1をパリティフラグとして用いる。



パリティ生成のアルゴリズムを図9.4を用いて簡単に説明する。D0にはビットデータd0~15が格納されている。まず、この内容を1ビット左にローテートしたものをD1に格納する。そして、D0とD1の内容をEORし、その結果をD0に格納する。D0に格納された値は、それぞれのビットデータの2ビット分のEORとなっている。次に、D0の内容を2ビット左にローテートしたものをD1に格納し、D0とD1の内容をEORしてから、その結果をD0に格納する。ここでD0に格納された値は、それぞれのビットデータの4ビット分のEORとなっている。さらに、D0の内容を4ビット左にローテート

トしたものを D1 に格納し、D0 と D1 の内容を EOR してから、その結果を D0 に格納する。このとき、D0 に格納された値は、それぞれのビットデータの 8 ビット分の EOR となっている。以下同様にして、16 ビット分の EOR を求める。データレジスタ D0 の各ビットは、それぞれすべてのビットデータを EOR した結果であり、そのため全ビット同じ値となる。このアルゴリズムに従ったプログラム例を次に示す。

MOVE	D0, D1	D0 → D1
ROL.W	#1, D1	D1 を左に 1 ビットローテート
EOR	D0, D1	$D1 \oplus D0 \rightarrow D1$
MOVE	D1, D0	D1 → D0
ROL.W	#2, D0	D0 を左に 2 ビットローテート
EOR	D0, D1	$D1 \oplus D0 \rightarrow D1$
MOVE	D1, D0	D1 → D0
ROL.W	#4, D0	D0 を左に 4 ビットローテート
EOR	D0, D1	$D1 \oplus D0 \rightarrow D1$
MOVE	D1, D0	D1 → D0
ROL.W	#8, D0	D0 を左に 8 ビットローテート
EOR	D0, D1	$D1 \oplus D0 \rightarrow D1$

## Ⅱ 68000 のサポート システム

## 20

## 68000 マイクロコンピュータ システム

マイクロコンピュータを利用するには、MPU、メモリ、クロック発生器などを個別に購入し、利用者が自らこれらをボード上に組み合わせて、マイクロコンピュータシステムを作ることもある。しかしより手軽にマイクロコンピュータを利用しようとする場合には、LSI メーカーやシステム メーカーが、広範囲な一般ユーザの利用を目的として発売しているマイクロコンピュータ ボードあるいはマイクロコンピュータ システムを用いると便利である。

この章では、以上のような目的の 68000 マイクロコンピュータ システムの概要を紹介する。詳細はメーカー発行のマニュアル類を参照されたい。

### 20.1 H 680 SBC

本節では、日立製作所から発売されている 16 ビット シングルボード コンピュータ シ

ステム (H 680 SBC) について概要を述べる。図 20.1 に H 680 SB 01 の外観写真を、図 20.2 に H 680 SBC のシステム構成図を示す。H 680 SBC システムの主要な機能モジュールとして、現在、次の 4 個が用意されている。(1) シングルボードのコンピュータ本体 (H 680 SB 01): MPU, ROM, RAM, PIA, ACIA, PTM (Programmable Timer Module) などを搭載。(2) モニタ ボード

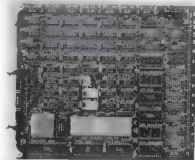


図 20.1 H 680 SB 01 の外観



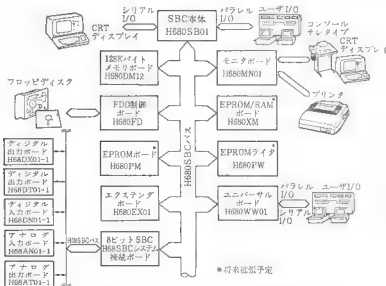


図 20.2 68000 マイクロコンピュータ システム (H 680 SBC システム)

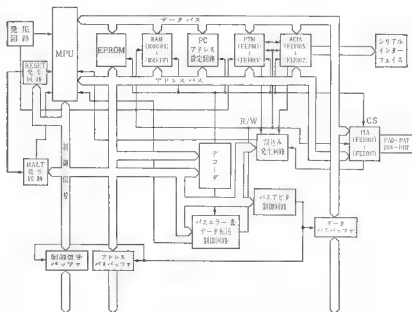
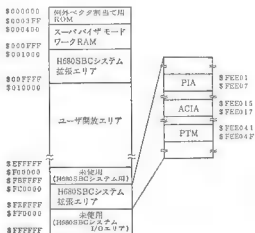


図 20.3 H 680 SB 01 のブロック図

(H680MN01): 標準 OS としてデバッグを実装し、RMS (Real time Monitor System) あるいはテキストエディタ、アセンブラなどが搭載できる。(3) RAM ボード (H680DM12): 128K バイトのダイナミックメモリを実装している。(4) フロッピディスク制御ボード (H680FD): フロッピディスクの制御を行う。これらのモジュールはコモンバス形式のインターフェイスをもっているため、ビルディングブロック形式で拡張することができる。

次に、H680SBC システムの核となるコンピュータ本体 H680SB01 について説明する。図 20.3 に H680SB01 のブロック図を示す。H680SB01 内のメモリは EPROM (最大 16K バイト)、RAM (4K バイト) を合せて最大 20K バイトのメモリを実装することが可能である。アドレスマップを図 20.4 に示す。システム用とユーザ用に別のエリアが割り当てられている。



■ 20.4 H680SB01 アドレスマップ

る。タイマは 6840 PTM を 1 個実装しており、2 本の 16 ビットタイマが使用できる。またボードへの外部割込みは 7 レベルまで入力することができる。

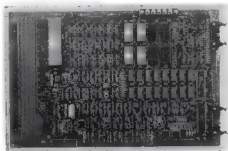
SBC を複数台用いてマルチプロセッサシステムを構成する場合、ボード上のジャンパコネクタをマルチモードに設定し、モニタボード (H680MN01) を用意すればよい。このときモニタボードに実装されているバスアービタによりバス権の制御が行われる。

パラレル I/O インターフェイス用には、6821 PIA が 1 個実装され、16 データラインと 3 制御線が使用可能である。シリアル I/O インターフェイス用には、6850 ACIA が 1 個実装され、RS-232C レベルインターフェイスをもつ機器に接続できる。転送レートは 300~9600 BPS (Bit Per Second) の間をプログラムにより 5 段階に設定でき

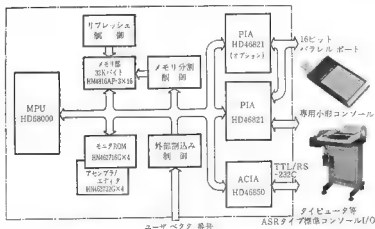
## 20.2 H 680 TR 01

H 680 TR 01 は、68000 を導入する場合の評価・学習用として開発されたワンボードコンピュータシステムである。図 20.5 にボードの外観写真を、図 20.6 にシステムブロック図を示す、以下に機能の概要を述べる。

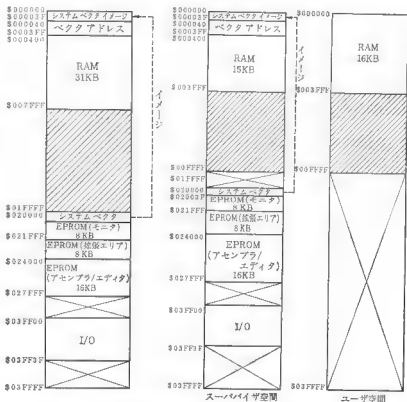
H 680 TR 01 は、32 K バイトのダイナミック RAM を標準実装し、さらにメモリ素子の差換えて、128 K バイトまでメモリ容量を拡張できる。また RAM 領域の全空間はシステムとユーザが共用できるモード（モードⅠ）と、スーパーバイザ空間とユーザ空間のそれぞれに分配されるモード（モードⅡ）で使用できる。これらの選択はプログラムでフラグを設定することによって行われる。各モードにおけるアドレスマップは図 20.7 に示すとおりである。システムプログラムは EPROM に書き込まれている。



■ 20.5 16ビットトレーニングモジュール  
(H 680 TR 01) の外観



■ 20.6 H 680 TR 01 のシステムブロック図



(a) モード I におけるアドレスマップ

(b) モード II におけるアドレスマップ

図 20.7 H680TR01 のアドレスマップ

図 20.6 に示したように、専用小形コンソール H68PC01 をパラレルポートにつないで、68000 のマシン語レベルの操作ができる。またオプションの PIA を 1 個追加すると 16 ビットバスのパラレル入出力ポートとなる。シリアルインターフェイスとして TTL または RS-232C レベルのインターフェイスを備えているので、ASR タイプの標準コンソールにつなぐことができる。転送レートは 300, 600, 1200, 2400, 4800, 9600 BPS のなかから選択できる。これにより紙テープベースでアセンブラ、テキストエディタを使用できる。

割込み機能としては、ボード内で発生する割込み（システム割込み）以外にユーザが外部から入力する割込み（ユーザ割込み）が使用できる。またパワー オンリセット、リセ

ットスイッチによるリセットおよびメモリの実装されていないアドレス空間をアクセスしたときのバスエラー検出機能を有している。

サポートソフトウェアとしては、入出力ルーチン、基数変換ルーチンなどのモニタ、アセンブラおよびテキストエディタが EPROM で用意されている。

なお、16 ピンの TTL の IC を約 30 個実装できるユーザ用のスペースを有しているため、小規模なハードウェアを実装して、システム機能の拡張を図ることができる。

マイクロコンピュータの応用にあたっては、開発するシステムの多様化、高度化に伴ってハードウェアとともにソフトウェアの効率的な開発が一層重要な鍵となる。しかしながらマイクロコンピュータ応用システムの開発では、

(1) ハードウェアが完成しないとソフトウェアのデバッグが難しく、両者の同時開発が困難である、

(2) マイクロコンピュータ応用のユーザシステムでは、通常、ソフトウェア開発のための言語プロセッサやデバッグ機能をもたないため、ユーザシステム上でのデバッグが困難である、

などの問題が伴い、効率的にシステム開発を行うことが難しい。

これらの問題を解決するために、マイクロコンピュータ開発支援装置がLSIメーカー、システムメーカーから種々発売されている。これによりプログラムの作成からデバッグ、さらにプロトタイプシステム上でのリアルタイムシミュレーションまで一貫したシステム開発を行うことができる。

本書では、種々の68000用開発支援装置のうち、特に日立製作所より発売されている68000マイクロコンピュータ開発支援システムH680SD300を紹介する。

## 21.1 H680SD300のハードウェアの概要

### 21.1.1 システム構成

図21.1はH680SD300の外観写真である。標準入出力機器として、コンソールCRT、プリンタおよびフロッピディスク装置が装備されている。本体部は、15枚のモジュールを収納でき、将来68120 IPCモジュールを追加すればコンソールCRT4台、プリンタ

1台がサポート可能となり、マルチステーションが実現できる。  
H680SD300 内部モジュールの構成は図 21.2 のように 4 種類 4 枚のモジュールからなり、各モジュール間は標準 VERSA バス<sup>†</sup>により結合している。表 21.1 に H680SD 300 の本体の仕様を示す。

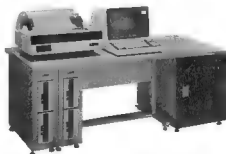


図 21.1 68000 マイクロコンピュータ開発支援システム (H 680 SD 300) の外観図

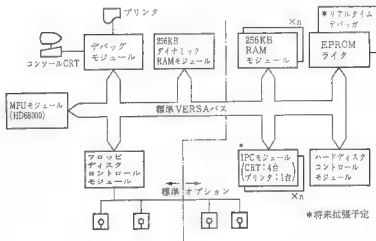


図 21.2 H 680 SD 300 のハードウェア構成図

### 21.1.2 システムの機能概要

H680SD300 は、標準システムとして、MPU モジュール、デバッグモジュール、フロッピーディスクコントロールモジュール、256 K バイトダイナミック RAM モジュールの 4 モジュールを装備している。これらは標準 VERSA バスで結合されるようになっている。

<sup>†</sup> 標準 VERSA バス：モトローラ社のマイクロコンピュータ開発支援システムで用いるモジュール間のバス方式

表 21.1 H 680 SB 300 の仕様

項 目	仕 様
プロセッサ 68000	データ : 1, 4, 8, 16, 32 ビット アドレス : 16 MB (24 ビット) アドレス形式 : 14 種類 レジスタ : 17 個 + 1 (ステータス レジスタ) ■ 命 令 : 56 種 (1, 2, 3, 4, 5 ワード)
外部割込み	7 レベル (同レベルの場合はスロット番号の小さい方の割込み優先)
バス制御	5 レベル (同レベルの場合はスロット番号の小さい方のバスリクエスト優先)
主記憶部容量	256 K バイト (標準)
入出力ポート	RS-232C 用ポート × 2 (標準), ポーレート 50~19200 ボー
プリンタ インターフェイス	並列ポート (セントロニクス仕様) ■
電 源	AC 100 V ± 10% 50/60 Hz
消費電力	標準構成時 (H 680 SD 300 本体, フロッピディスク, CRT ディスプレイ, プリンタ) の合計で 1100 VA
動作温度範囲	10~35°C
寸 法	高さ 650 × 幅 1540 × 奥行 800 [mm]

**a. MPU モジュール** MPU モジュール上には, MPU (68000), クロック発生回路, 4 セグメントのメモリ管理ユニット, プライマリ/セカンダリ マップ切換えロジック (次節で説明) およびモジュールの自己診断用のファームウェアがある。メモリ管理ユニットにより, オペレーティングシステム (OS) の管理下でのメモリ割当てや, マルチタスキングが可能となる。またリアルタイム マルチタスキング OS によるタスクの同時処理により, プログラムの開発をよりスピードアップさせることができる。たとえばアセンブラでプリンタを使用中に, コンソール CRT でプログラムの編集が行える。

**b. デバッグ モジュール** デバッグ モジュールは, デバッグ機能をもつファームウェア, バス管理ロジック, プリンタ用入出力ポート, コンソール CRT 用 RS-232C ポートおよびホスト コンピュータとの交信用 RS-232C のポートから構成されている。

**c. フロッピディスク コントロール モジュール** フロッピディスク コントロール モジュールには, 68000 システムからのデータ要求の処理と, フロッピディスク用の自己診断のために, 専用マイクロコンピュータが使用されている。このようなマルチプロセッシング技術を使用することにより, システムのパフォーマンスを向上させ, より効果的なシステムの利用を図っている。

**d. 256 K バイト ダイナミック メモリ モジュール** メモリには 64 K ビットのダイナミック RAM チップを使用し, 65536 ワード (16 ビット/ワード) のブロック 2 個から構成されている。ダイナミック RAM のリフレッシュのためにモジュール内にリフレッシュ機能をもっている。8 ビットごとに 1 ビットのパリティ ビットを付加して, パリティ



エラーチェックを行っている。パリティエラーが発生したときは、MPU に対してバスエラーを伝える。メモリアドレスの割当てはボード上のスイッチで任意に設定可能である。またこのメモリモジュールは、増設可能で、1M バイト以上の常駐メモリを使用できる。

e. **H 680 SD 300 周辺装置** H 680 SD 300 で使用する周辺装置の仕様の概要を表 21.2 に示す。

表 21.2 H 680 SD 300 周辺装置

入出力装置名	機能仕様
コンソール CRT	<ul style="list-style-type: none"> <li>◦ インテリジェント機能付</li> <li>◦ 1920 文字/画面</li> <li>◦ 14 インチ画面、緑色文字</li> </ul>
プリンタ	<ul style="list-style-type: none"> <li>◦ 両方向印字形シリアルプリンタ</li> <li>◦ ドットインパクト方式</li> <li>◦ 180 文字/sec</li> <li>◦ 132 文字/行</li> </ul>
フロッピーディスクドライブ装置	<ul style="list-style-type: none"> <li>◦ 両面単密度使用 (日立製 FDD 402)</li> <li>◦ 500 K バイト/台</li> <li>◦ IBM 3740 フォーマット</li> </ul>
高速ラインプリンタ (オプション、将来)	<ul style="list-style-type: none"> <li>◦ 700 行/min</li> <li>◦ 132 文字/min</li> <li>◦ 横字インパクト方式</li> </ul>

### 21.1.3 システムの拡張性

a. **マルチステーション H 680 SD 300** は同時に複数のユーザがプログラム開発を行えるマルチステーションに拡張可能である。IPC モジュール 1 枚の追加で 4 台のコンソール CRT と 1 台のプリンタを接続できる。この場合メモリを追加することが必要である。

b. **リアルタイムデバッグ装置**  
68000 ASE (Adaptive System Eva-

luator) モジュールおよび ASE BOX を追加することにより、リアルタイムで 68000 ユーザシステムをデバッグすることができる。ASE BOX 内の 68000 は、ユーザシステムの MPU として動作し、コンソール CRT 上のキーにより簡単にデバッグできる。

c. **EPROM ライタ** EPROM ライタモジュールを追加することにより、EPROM に対し、書き込み、読み出し、照合、ブランクチェック、コピーができる。

d. **ハードディスク装置** ハードディスク装置を外部記憶装置として接続可能である。フロッピーディスク装置に比べ、高速処理ができ、多くのプログラムおよび大規模なプログラムの開発が容易となる。

## 21.2 ソフトウェア構成

図 21.3 に H 680 SD 300 のソフトウェアの構成を示す。FDOS, EMS (Executive Monitor System), CRT エディタ, マクロアセンブラ, リンケージエディタ, FORTRAN コンパイラ, PASCAL コンパイラ, スーパー PL/H コンパイラが提供されており、さらにシ

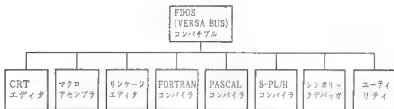
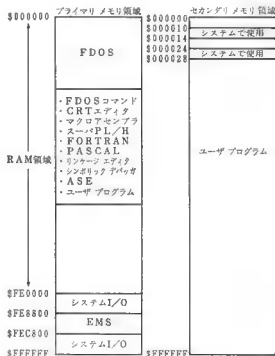


図 21.3 H 680 SD 300 のソフトウェア構成図

シンボリック デバッガなどのオプション機能の開発が進められている。これら各ソフトウェアの内容については、22章で述べる。

H 680 SD 300 を使用して、FDOS、EMS のもとでユーザプログラムをデバッグする際、プライマリ メモリマップとセカンダリ メモリマップの概念が必要となる。ここでは図 21.4 に示す H 680 SD 300 のメモリマップを用いて上記各概念について説明する。



注 1) H 680 SD 300 上のメモリボードのスイッチを切り換えることにより、メモリを 64 K バイト単位にプライマリメモリ領域またはセカンダリメモリ領域に割り当てることができる。

注 2) RAM は 256 KB 単位に増設可能である。

図 21.4 H 680 SD 300 メモリマップ

**a. プライマリ メモリマップ** プライマリ メモリマップの場合、ユーザプログラムは FDOS と同一領域にロードされ、ユーザモードとして実行させる。したがってデバッグ対象のプログラムは、ユーザモードでなければならない。この場合、FDOS のデバッグコマンドを使用して、ユーザプログラムをデバッグする。

**b. セカンダリ メモリマップ** セカンダリ メモリマップの場合、FDOS の「SMLOAD」コマンドにより、フロッピディスク上のユーザプログラムがセカンダリメモリマップ上にロードされる。この場合、ユーザプログラムはスーパーバイザモードおよびユーザモードの両方を使用することができる。さらに 16 M バイトの全メモリ空間をユーザが使用することが可能である。ただし (000010)<sub>16</sub> 番地および (000024)<sub>16</sub> 番地からの各 4 バイトはシステムで使用し、ユーザは使用できない。セカンダリメモリマップの場合、EMS のデバッグコマンドを使用してユーザプログラムをデバッグできる。

## 21.3 H 680 SD 300 の使い方

### 21.3.1 入出力機器の指定

H 680 SD 300 は入出力機器としてフロッピディスク、コンソール CRT およびプリンタを備えている。これらの入出力機器の指定方法は本システムを使いこなす上で重要な点である。本システムの FDOS では、入出力機器の指定を次のように行う。

#### a. デバイスの指定方法

##### (1) フロッピディスク

#FD×× (××はドライブ装置の番号 00, 01, 02 および 03)

##### (2) CRT ディスプレイ

#CN×× (××: 00 および 01)

##### (3) ログオンターミナル

#

**b. ファイルの指定方法** FDOS におけるファイル名の標準フォーマットは次のとおりである。

<ボリューム名>: <ユーザ番号>, <カタログ名>, <ファイル名>, <拡張名>, <保護キー>
---

(1) ボリューム名: フロッピディスクにつけた固有名称 (1~4 桁の英数字)

(2) ユーザ番号: ユーザにつけられた固有番号 (1~9999 の 10 進数)

- (3) カタログ名 : ファイルを管理するための名前 (1~8 桁の英数字)
- (4) ファイル名 : 最も基本となるファイル名 (1~8 桁の英数字)
- (5) 拡張名 : ファイルの種類を示す記号 (2 桁の英字)

システムで使用している拡張名は次のとおりである。

LO: ロード モジュール ファイル  
 RO: リロケートブルオブジェクト ファイル  
 PA: PASCAL ソース ファイル  
 FT: FORTRAN ソース ファイル  
 AL: アセンブラ ソース ファイル  
 SY: FDOS システム ファイル  
 CF: コマンド ファイル  
 LS: 言語プロセッサのリスト出力ファイル  
 PC: PASCAL 中間語ファイル  
 LL: リンケージエディタのリスト出力ファイル

- (6) 保護キー : ファイルの書き込みに対する保護キー (2 桁の数字)

### 21.3.2 システムの動作モード

H680SD300 のソフトウェアには次の 2 つのモードがあり、それぞれ異なった方法で起動される。

- (1) FDOS モード
- (2) EMS モード

FDOS モードでは FDOS システム ディスクが必要である。H680SD300 では図 21.3 に示すようなソフトウェアが 5 枚のシステム ディスクに格納されている。必要に応じて、各ディスクを使用することができる。EMS モードでは FDOS システム ディスクは不要である。上記 2 つのモードの起動方法、停止方法については、本書では省略する。メーカーから出版されているマニュアルを参考にされたい。

### 21.3.3 ソフトウェアの開発手順

H680SD300を使用したシステム開発手順を図 21.5 に示す。

- a. ソフトウェアの作成 ソフトウェアの作成は大きく次の 3 ステップで行われる。

- (1) ソース プログラム ファイルの作成
- (2) アセンブラおよびコンパイラの実行 アセンブラ, PASCAL, FORTRAN あるいはスーパー PL/H コンパイラによりオブジェクト モジュールを作成する。

- (3) ロード モジュールの作成 リンケージエディタを使用して、アセンブラやコ

ンバイラが出力したオブジェクトモジュールを絶対番地形式のロードモジュールに変換する。

### b. デバッグの方法 H 680 SD 300

には次の3つのデバッグ機能がある。

(1) EMS によるソフトウェアデバッグ機能 EMS にはユーザプログラムのロード、メモリ内容の表示/変更およびユーザプログラムの実行等のデバッグ機能がある。EMS の下では、ユーザプログラムはプライマリメモリマップまたはセカンダリメモリマップ上の絶対番地のアドレスにロードされる。

(2) シンボリックデバッグによるソフトウェアデバッグ機能 シンボリックデバッグはFDOS下で稼動するデバッグで、ユーザプログラムのロード、メモリ内容の表示/変更およびユーザプログラムの実行等のデバッグ機能がある。シンボリックデバッグの下ではユーザプログラムはプライマリメモリマップ(後にセカンダリメモリマップもサポートする予定である)上の論理アドレスにロードされる。また将来は PA

SCAL 等の高級言語のシンボルによるデバッグ機能も可能となる。

(3) ASE によるソフトウェア/ハードウェアデバッグ機能 ASE には H 680 SD 300 に接続したユーザシステムのハードウェアおよびソフトウェアの両方を同時にデバッグする機能がある。

その特徴は次のとおりである。

- i) ユーザシステムのクロックで動作し、リアルタイムでソフトウェア、ハードウェア両方のデバッグが可能である。

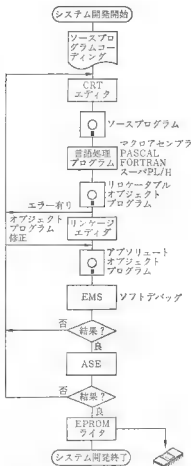


図 21.5 H 680 SD 300 によるシステム開発手順

- ii) 2箇所ブレークポイントをハードウェアで設定できるので、リアルタイムブレークが可能である。
- iii) 125 ns ごと的高速トレース、MPU インターフェイスチェック、サイタルリセットによる解析機能、タイムアウトチェック機能など多くのハードウェアチェック機能をもっている。

アプリケーションプログラムの効率良い開発を行うためには、サポートソフトウェアが充実していることが重要である。一例として、図 22.1 に日立製作所の 68000 用サポートソフトウェアを示す。レジデントシステムとしては、シングルボードコンピュータ (SBC) 用と開発支援システム H 680 SD 300 用のものがある。またクロスシステムとして、大形計算機 HITAC M シリーズおよび IBM 370 用のものがある。モトローラ社のサポートソフトウェアシステムもほぼ同様である。サポートソフトウェアの詳細は、メーカーから出されている説明書を参考にされたい。本書では、レジデントのなかから、いくつかのソフトウェアについての概要を紹介しておく。



図 22.1 68000 用サポートソフトウェア

## 22.1 FDOS (Floppy Disk Operating System)

FDOS は、大容量で、操作性のよい両面フロッピディスクを記憶媒体としたタスク管理機能、ジョブ管理機能およびファイル管理機能を有し、スーパーPL/H、FORTRAN、PASCALなどの言語プロセッサ、ユーティリティならびにユーザプログラムを統一して管理する。またハードディスクベースでマルチプログラミングをサポートする機能も具備している。その特長として、

- (1) コマンドインタプリタ方式の採用により、会話形式でオペレーションができる。また、よく用いる定型的なコマンド群を登録することにより、操作性を向上させることができる。
- (2) プログラミングからデバッグまで、一貫してフロッピディスクベースでできる。最大4台(3Mバイト)までのフロッピディスク(両面または片面単密度)をサポートする。
- (3) CRTエディタ、マクロアセンブラ、スーパーPL/H、FORTRAN、PASCALリネージエディタ、ASE、EPROMライターおよびFDOSユーティリティなどのプログラムを簡単に呼び出せる。

表 22.1 FDOS 内部コマンド

項番	分類	コマンド	機能
1	セッション管理	(BREAK キー)	セッションを開始する
2		LOGOFF	セッションを終了する
3		BYE	セッションを終了する
4	ジョブ管理	CHAIN	カタログプロセスの実行
5		RETRY	アボートした現コマンドを再実行させる
6		PROCEED	アボートした次のコマンドから実行させる
7		END	カタログプロセスによる実行を終了させる
8		BATCH	バックグラウンドでのカタログプロセスの実行
9		CANCEL	バックジョブの中止
10		QUERY	バックジョブの状況表示
11		ARGUMENTS	CHAIN および BATCH コマンド用パラメータを作成する
12		NOARGUMENTS	CHAIN および BATCH コマンド用パラメータをリセットする
13	プログラム実行	LOAD	プログラムをメモリにロードする
14		START	プログラムを実行させる
15		CONTINUE	中断したプログラムを実行させる
16		STOP	プログラムの実行を中断させる
17		TERMINATE	プログラムの実行を止める
18		BSTOP	ブレークキーにより、プログラムを中断させる
19		BTERM	ブレークキーにより、プログラムを止める
20	ディフォルト管理	USEID	当該セッションのディフォルトボリュームID、ユーザ番号の変更
21		DEFAULTS	当該セッションのディフォルトボリュームID、ユーザ番号の表示
22	時刻管理	DATE	現在日時を表示し、日付を変更する (ユーザ番号のみ)
23		TIME	現在日時を表示し、時刻を変更する (ユーザ番号のみ)



表 22.2 FDOS ユーティリティコマンド

項番	分類	コマンド	説明
1	ファイル 管理	RENAME	ファイル名の変更
2		DEL	ファイルの削除
3		DIR	ディレクトリの内容表示
4		FREE	ディスク空き容量の表示
5	インストール	INIT	ディスクシステムテーブルの初期化
6	ユーティ リティ	BACKUP	ディスクコピー/ベリファイ
7		COPY	ファイルのコピー/ベリファイ
8		CONV	M/IBM フォーマットのファイルをSD 800 用に変更する
9		DUMP	ディスク/ファイル内容の表示
10		LIST	ソースファイルの表示
11		MLM	S タイプオブジェクトをバイナリファイルに変換する
12	デバッグ 機能	PATCH	ロードモジュールの修正
13		DEBUG	FDOS デバッグコマンドへ制御を渡す
14		SMLOAD	セカンダリメモリマップ上にオブジェクトをロードする
15	プログラミ ングシステム	E	CRT ユティリティへ制御を渡す
16		ASM	マクロアセンブラへ制御を渡す
17		PLH	スーパーPL/HI へ制御を渡す
18		FORTTRAN	FORTTRAN へ制御を渡す
19		PASCAL	PASCAL フェーズ1へ制御を渡す
20		PASCAL2	PASCAL フェーズ2へ制御を渡す
21		LINK	リンケージエディタへ制御を渡す

表 22.3 DEBUG サブコマンド

項番	分類	コマンド	説明
1	ブレーク ポイント	BR	ブレークポイント番地の指定、表示
2		NOBR	ブレークポイント番地の解除
3	トレース および 条件ブレーク	OP	オプション指定 (1) トレース (2) 特定アドレスのメモリ内容の一覧/変化によるブレーク (3) 指定命令実行後のブレーク
4		VA	上記ブレーク条件 (2) の比較値指定、表示
5		VM	VA で指定した値のマスクビットパターンの指定、表示
6		VL	上記ブレーク条件 (2) のメモリアドレスの指定、表示
7		MC	上記ブレーク条件 (3) の命令数指定、表示
8		XM	例外発生時のブレークマスク指定、表示
9	プログラム 実行	G	現在の PC 番地からプログラムを実行
10	メモリ、レ ジスタ内容 の表示変更	MD	指定番地からのメモリ内容を指定回数表示
11		MM	指定番地のメモリ内容を変更
12		DF	全レジスタ内容表示
13		Ri	レジスタ内容の表示、変更
14	プリンタ 出力	AP	プリンタへ接続
15		MOP	指定番地からのメモリ内容を指定回数プリンタへ出力
16		DP	プリンタ切断
17	その他	EXIT	FDOS へ制御を移行 (ただしタスクは実行)
18		QUIT	全てを終了し FDOS へ制御を移行
19		ST	タスクのステータスを表示

- (4) セクタ単位の効率的なスペース管理を行う。
- (5) ファイルをユーザ別に保護・管理する。
- (6) ユーザプログラムを簡単に呼び出せる。
- (7) マルチタスク処理を行い、スループットの向上を図っている。
- (8) MMU (Memory Management Unit) を使用して、タスク対応に独立した論理空間を与え、メモリの保護を実現している。

表 22.1、表 22.2、表 22.3 に FDOS の制御コマンド、FDOS 下のファイルを管理するユーティリティコマンド、ユーザプログラムのデバッグに用いる DEBUG サブコマンドのそれぞれを示す。

## 22.2 EMS (Executive Monitor System)

EMS は物理空間用のモニタで、プライマリマップおよびセカンダリマップ上でのプログラムのロードやデバッグを行う。また FDOS のブートストラップやプリンタへの出力機能などをもっている。

EMS の主な機能は次のようなものである。

- (1) 紙テープ上のオブジェクトプログラムのロード、メモリ内容のパンチおよび照合
- (2) アドレスストップの指定、解除および表示
- (3) ブレークポイントの指定、解除および表示
- (4) プログラムの実行、トレース
- (5) メモリ内容の表示、変更
- (6) レジスタ内容の表示、変更
- (7) プリンタへのハードコピー出力
- (8) プライマリマップおよびセカンダリマップの切換え
- (9) メモリのテスト
- (10) FDOS のブートストラップ

## 22.3 CRT エディタ

CRT エディタは、オペレーティングシステムの管理下で動作し、ソースプログラムの作成や変更を CRT 画面上で行う。この CRT エディタはコマンドモードおよびページモ

ードの2つのモードをもち、専用CRTコンソールのカーソル機能、制御用キャラクタおよびファンクションキーを用いて、ソースファイル内の文字や行の挿入、変更および削除を行う。CRTエディタの機器構成を図22.2に示す。機能の概要は次のとおりである。

(1) ページモード

- (i) 文字の変更、削除、 (ii) 行の挿入、削除、 (iii) 1行スクロールアップ/スクロールダウン
- (iv) 1ページスクロールアップ/スクロールダウン

(2) コマンドモード

- (i) 文字列のサーチ、 (ii) 文字列の変更

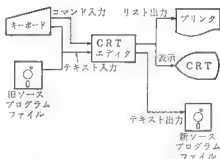


図 22.2 CRT エディタの機器構成

## 22.3 リンケージエディタ

リンケージエディタは、別々にコンパイルおよびアセンブルされた複数のリロケータブルオブジェクトモジュールを結合し、1つのロードモジュールまたは1つのリロケータブルオブジェクトモジュールをつくる。このときセグメントの属性の決定、アドレス空間の計算、ライブラリの検索、外部参照の解決、オブジェクトモジュールの再配置およびエラーメッセージの出力を行う。さらにモジュールマップ、外部定義シンボルのテーブル、未定義または2重定義シンボルなどの情報もプリント出力する。

リンケージエディタの基本的な処理は、次の2つである。

a. 外部参照関係の解決 別

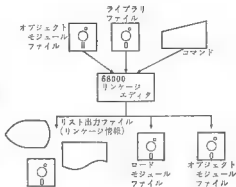


図 22.3 リンケージエディタの入出力機器構成

別にコンパイルまたはアセンブルしたプログラム群の間で、別のプログラム中のシンボルを参照（これを外部参照とよぶ）することを可能とする。

**b. モジュールの結合** 別々にコンパイルまたはアセンブルしたプログラム群を、1本のロードモジュールに編集、結合する。

図22.3にリンケージエディタの入出力関連図を示す。

## 22.5 マクロアセンブラ

H680SD300のFDOSの管理下で動作するマクロアセンブラの機器構成を図22.4に示す。68000アセンブリ言語で記述されたソースプログラムをフロッピディスクより入力して、機械語に翻訳し、絶対配置（アブソリュート）または再配置可能（リロケートブル）なオブジェクトモジュールに編集して出力する。

主な特長は次のとおりである。

- (1) マクロ機能により、コーディングが簡略化される。
- (2) 条件付アセンブリングにより1つのプログラムを多目的に利用できる。
- (3) リロケートブルなオブジェクトの作成が可能のため、1つのプログラムをモジュール単位に分けてアセンブルし、リンケージエディタにより結合および絶対番地の割当てをすることができる。

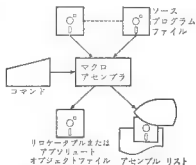


図 22.4 マクロアセンブラの機器構成

- (4) リストおよびオブジェクトはアセンブラ制御命令によって制御できる。
- (5) 標準登録シンボル数：2000個
- (6) 標準アセンブルステップ数：8000ステップ

マクロアセンブラのコマンド、制御命令の詳細は、メーカーから出版されているマニュアルを参照されたい。

## 22.6 FORTRAN

68000用のFORTRANは、FORTRAN77のサブセットに、ビット処理機能を追加し

た高級言語である。このコンパイラは、ソースプログラムをリロケータブルなオブジェクトモジュールに変換する。データのタイプは、整数形（2バイト、4バイト）、実数形（4バイト）、論理形および文字形がある、それぞれ変数または配列として使用できる。一方メモリ内のデータをファイルとして扱える内部ファイル機能があり、印字するデータの編集が容易になっている。また構造化プログラミングがしやすい機能や三角関数、双曲線関数で代表される多くの組込み関数等が備えられている。さらにビット処理を行うサブルーチンもある。

## 22.7 PASCAL

PASCAL は、N. Wirth によって設計された高級言語である。系統的にプログラムを設計したり、アルゴリズムを記述したりするのに適している。その特徴は、データタイプとして配列、レコード、集合、ファイル、ポインタ形をもつなど、論理的によく整理されたものとなっている点である。また条件文や繰返し文の種類にみられるように、洗練された制御構造は、構造化プログラミングに適したものとなっている。

68000 用の PASCAL は、いわゆる標準 PASCAL とよばれる仕様（UCSD 仕様）に、68000 アーキテクチャの特長を十分発揮できる特有な機能を追加している。

言語機能の概要は次のとおりである。

(1) 13 種類の形定義が使用できる： 符号付整数形 4 バイト（ロングワード）、符号付整数形 2 バイト（ワード）、符号付整数形 1 バイト（バイト）、部分範囲形、スカラー形、配列形、レコード形、集合形、ファイル形、文字列形、ポインタ形、文字形、論理形

(2) 変数は、上記一般形を宣言でき、またプログラム自身が自由に構成することも可能である。

(3) 演算には、NOT 演算記号、乗法演算記号（\*、DIV、MOD、AND）、加法演算記号（+、-、OR）、関係演算記号（=、<>、<、>、<=、>=、IN）が使用できる。

(4) 代入文、手続き文、GOTO 文、空文、EXIT 文、複合文、WITH 文、IF 文、CASE 文、繰返し文（WHILE 文、REPEAT 文、FOR 文）、入出力文等の実行文が使用できる。

(5) 手続きには、関数形手続きと、サブルーチン形手続きとがあり、各手続きは、再帰的呼出し可能（RECURSIVE）で、FORWARD 属性を使用することで、手続きをコンパイル単位の外に拡張することもできる。

(6) プログラム中で宣言するすべての変数、手続き、ラベルは、その有効範囲をもつことができる。

PASCAL の概略を次のプログラム例を用いて説明する。

```

Program Call (input, output);           ①
  var i, m, n: integer;                 ②
  procedure cube;                       ③
  begin n := m * m * m end;             ④
begin for, i := 1 to 10 do               ⑤
  begin read (m); cube; writeln (n); end; ⑥
end.                                     ⑦

```

- ① プログラム頭部；識別名 Program から始まり、プログラム名 Call、プログラムパラメータ名 input, output と続く。input, output は基準ファイルとよばれる。
- ② 変数宣言部；ローカルに変数を定義する。この例では、変数名 i, m, n に対して整数形を定義している。
- ③ 手続き宣言部の頭部；手続き名と引数（オプション）を指定する。プログラム名 Call の実行文 ⑤ 以前に使用する手続きがすべて宣言されていなければならない。
- ④ 手続き cube の実行文部；複合文の形（③で説明）をとり、 $m^3$  の値を n に代入している。
- ⑤ プログラム名 Call の実行文部；begin で始まり ⑦ の end で閉じている。この形を複合文とよび、開かれた各実行文が、1つのまとまりをもつ。第1実行文の for 文は次の実行文を10回繰り返す役割をする。この例では、該当実行文が複合文の形をしているので、全体が10回繰り返される。
- ⑥ read (m) は、基準ファイル input から、変数 m に整数を読み込む。読み込んだ値を手続き cube を引用して3乗値を n に求める。求めた n を writeln (n) によって基準ファイル output へ出力する。
- ⑦ 最も外側の複合文（第1レベル）を終了するときには end にピリオドを付記する。

## 22.8 スーパ PL/H

スーパ PL/H は汎用高級言語 PL/I からマイクロコンピュータの応用システム記述に必要な機能を抜粋し、さらにマイクロコンピュータ特有の機能を付加して構成した言語である。また8ビットマイクロコンピュータ 6800 用の PL/H との上位互換。インテル社

の 16 ビット マイクロコンピュータ 8086 用の PL/M86 との互換性も考慮されている。さらにフォーマブル機能<sup>1</sup>、3 次元配列、3 レベル構造体なども可能である。

PL/H は、アセンブラ言語に比べプログラムが読みやすく、少ないステップ数で記述できるため、プログラムの生産性と品質の向上を期待できる。

ソースプログラムファイルから入力したスーパ PL/H のプログラムは文とよばれるプログラムを記述する要素の集まりで構成される。これらの文はスーパ PL/H コンパイラによって、記憶数を確保したり、プログラムの機能を実現するための機械語の列に翻訳され、オブジェクトモジュールファイルに格納される。

スーパ PL/H 言語の機能概要は次のとおりである。

- (1) 7 種類のデータ形を使用できる 符号なし整数形 1 バイト (バイト)、符号なし整数形 2 バイト (ワード)、符号付整数形 2 バイト、符号付整数形 4 バイト、実数形、ビット形、ポインタ形
  - (2) 変数はスカラ変数、3 次元までの配列、3 レベルまでの構造体として宣言できる。
  - (3) 演算は算術演算 (+, -, \*, /), 比較演算 (<, =, >=, <=, >, <), 論理演算 (NOT, AND, OR, XOR), 番地参照演算 (@) が可能である。
  - (4) 代入文, CALL 文, RETURN 文, GOTO 文, IF 文, 単純 DO 文, 反復 DO 文, DO-WHILE 文, DO CASE 文, 機械依存文 (HALT, ENABLE, DISABLE, TESTANDSET) 等の実行文が使用できる。
  - (5) 手続きには関数形手続きとサブルーチン手続きとがあり、再入可能属性 (REENTRANT) を指定することができる。さらにサブルーチン形手続きには、割り込み属性 (INTERRUPT) を指定することができる。
  - (6) ブロックの中で宣言するすべての変数、手続き、ラベル、マクロは有効範囲をもつことができる。
  - (7) 外部名定義属性 (PUBLIC) と外部名参照属性 (EXTERNAL) を用いて変数、および手続きの有効範囲をコンパイル単位の外に拡張できる。
  - (8) 組込み関数、組込み手続、組込み変数を使用できる。
  - (9) マクロ、INCLUDE ライブラリを使用できる。
- スーパ PL/H の概略を次のプログラム例を用いて説明する。

<sup>1</sup> ロードモジュール (たとえば ROM 化したプログラム) をアドレス空間のどこに割り当ててもよいという機能

```

EX1:DO;                                ①
/* FOUR RULES OF ARITHMETIC */        ②
DECLARE(A,B,C,D,E,F)WORD;             ③

A=25; /* A AND B ARE NON-NEGATIVE */   ④
B= 8; /* INTEGER LESS THAN 65536 */    ⑤
C=A+B;                                  ⑥
D=A-B;                                  ⑦
E=A*B;                                  ⑧
F=A/B;                                  ⑨
END;                                     ⑩

```

- ① プログラムは EX1: のように 〈プログラム名〉: (コロン) で始まり、次に DO; が続く。この DO; と最後の END; の間がプログラムである。1行目のような行を単純 DO 文といい、END; まですべてを一般に単純 DO ブロックとよぶ。
- ② /\* と \*/ で囲まれた部分を注釈という。コンパイラがプログラムを機械語に翻訳するときは空白として処理する。
- ③ プログラムの中で使用する 6 個の変数を宣言している。このような文を宣言文という。宣言文は記憶場所を確保するためのもので、その変数が使用される前に宣言しておく必要がある。A から F までの変数は WORD 形であることを示し、それぞれに 2 バイト (16 ビット) ずつの記憶領域が割り当てられる。
- ④ 空白行。プログラムを読みやすくするために挿入してある。
- ⑤ 代入文。右辺の値を左辺 A に代入する。DO 文や宣言文と同様に、代入文も ; (セミコロン) で終る。この行は代入文の後に注釈をとまっている。
- ⑥ 変数 B に値 8 を代入する。
- ⑦ A の値に B の値を加え、その結果の値を C に代入する。
- ⑧ A の値から B の値を引き、その結果の値を D に代入する。
- ⑨ A の値と B の値を掛け、その結果の値を E に代入する。乗算の結果は上位ワードを切り捨てた値となる。
- ⑩ A の値を B の値で割り、その結果の値を F に代入する。除算の結果は小数点以下を切り捨てた値になる。
- ⑪ END 文。1 行目の DO 文と対応してプログラムの範囲を表す。



## 付 録

付録として、258～261 ページに実行命令オペランド一覧表を、262～266 ページにオペレーションコード一覧を掲げる。実効アドレス形式については表 4.1 を参照されたい(45 ページ)。なお、オペレーションコード一覧表に用いられる用語の定義は以下のとおりである。

サ イ ズ	0 0	バイト
	0 1	ワード
	1 0	ロングワード
R/M	(レジスタ/メモリ): レジスタ=レジスタ=0, メモリ=メモリ=1	
dr	(direction): 右シフト=0, 左シフト=1	
i/r	(カウントソース): イミディエイト カウンタ=0, レジスタ カウンタ=1	







ステーションオペランド										データサイズ				コンディションコード							備 考	
Am	Am	Am	Am	Am	Am	Am	Am	Am	Am	B	W	L	X	N	Z	V	C					
○	○	○	○	○	○					○	○	○	—	*	*	0	0					
										○	○	○	*	*	*	*	*	(注) 下位 8 ビットのみ対応				
○	○	○	○	○	○					○	○	○	*	*	*	*	*	特種命令				
													—	—	—	—	—					
												○	—	—	—	—	—	特種命令				
○		○	○	○	○							○	○	—	—	—	—					
レ	ジ	ス	タ	リ	ス	ト						○	○	—	—	—	—					
												○	○	—	—	—	—					
												○	○	—	—	—	—					
												○	—	*	*	0	0					
												○	—	*	*	0	0					
○	○	○	○	○	○					○			*	U	*	U	*					
○	○	○	○	○	○					○	○	○	*	*	*	*	*					
○	○	○	○	○	○					○	○	○	*	*	*	*	*					
												○	—	—	—	—	—					
○	○	○	○	○	○					○	○	○	—	*	*	0	0					
○	○	○	○	○	○					○	○	○	—	*	*	0	0					
○										○	○	○	—	*	*	0	0	(注) ワードサイズの場合、特種命令				
												○	—	—	—	—	—	特種命令				
												○	○	○	—	*	*	0	*			
○	○	○	○	○	○							○	○	○	*	*	*	0	*			
○	○	○	○	○	○							○	○	○	*	*	*	0	*			
												○	○	○	*	*	*	0	*			
												○	○	○	*	*	*	0	*			
												○	○	○	*	*	*	0	*			
												○	○	○	*	*	*	0	*			
												○	○	○	*	*	*	0	*			
												○	○	○	*	*	*	0	*			
												○	○	○	*	*	*	0	*			
												○	○	○	*	*	*	0	*			
												○	○	○	*	*	*	0	*			
												○	○	○	*	*	*	0	*			
												○	○	○	*	*	*	0	*			
												○	○	○	*	*	*	0	*			
												○	○	○	*	*	*	0	*			
												○	○	○	*	*	*	0	*			
												○	○	○	*	*	*	0	*			
												○	○	○	*	*	*	0	*			
												○	○	○	*	*	*	0	*			
												○	○	○	*	*	*	0	*			
												○	○	○	*	*	*	0	*			
												○	○	○	*	*	*	0	*			
												○	○	○	*	*	*	0	*			
												○	○	○	*	*	*	0	*			
												○	○	○	*	*	*	0	*			
												○	○	○	*	*	*	0	*			
												○	○	○	*	*	*	0	*			
												○	○	○	*	*	*	0	*			
												○	○	○	*	*	*	0	*			
												○	○	○	*	*	*	0	*			
												○	○	○	*	*	*	0	*			
												○	○	○	*	*	*	0	*			
												○	○	○	*	*	*	0	*			
												○	○	○	*	*	*	0	*			
												○	○	○	*	*	*	0	*			
												○	○	○	*	*	*	0	*			

ABCD	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	1	0	0	Destination Register				1	0	0	0	0	R/M	Source Register	

R/M (register/memory): register — register = 0, memory — memory = 1

ADD	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	1	0	1	Register			Op-Mode			Effective Address					
	Op-Mode															
	B	W	L													
	000	001	010	$Dn + EA \rightarrow Dn$												
	100	101	110	$EA + Dn \rightarrow EA$												
	—	011	111	$An + EA \rightarrow An$												

ADD Immediate	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	0	0	0	1	1	0	Size			Effective Address				

ADDQ	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1	0	1	Register			0	Size			Effective Address				

ADDX	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	1	0	1	Destination Register				1	Size		0	0	R/M	Source Register	

R/M (register/memory): register — register = 0, memory — memory = 1

AND

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	Register				Op-Mode			Effective Address				

Op-Mode

B	W	L	
000	001	010	$Dn \wedge EA \rightarrow Dn$
100	101	110	$EA \wedge Dn \rightarrow EA$

AND Immediate	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	0	0	0	0	1	0	Size			Effective Address				

Bcc	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1	1	0	Condition						8 bit Displacement					

BIT 操作 Dynamic	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	0	0	Register			1	Type			Effective Address				

BIT 操作 Static	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	0	0	1	0	0	0	Type			Effective Address				

Bit Type Codes: TST = 00, CHG = 01, CLR = 10, SET = 11

BSR	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1	1	0	0	0	0	1	8 bit Displacement							

CHK	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1	0	0	Register			1	1	0	Effective Address					

CLR	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1	0	0	0	0	1	0	Size			Effective Address				

CMP	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	1 0 1 1	Register	Op-Mode	Effective Address
	Op-Mode B W L 000 001 010 011 EA — 011 111 ARI-EA				
CMP Immediate	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	0 0 0 0	1 1 0 0	Size	Effective Address
CMPM	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	1 0 1 1	Register	1 Size	0 0 1 Register
DBcc	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	0 1 0 1	Condition	1 1 0 0	1 Register
DIVS	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	1 0 0 0	Register	1 1 1	Effective Address
DIVU	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	1 0 0 0	Register	0 1 1	Effective Address
EOR	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	1 0 1 1	Register	1 Size	Effective Address
EOR Immediate	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	0 0 0 0	1 0 1 0	Size	Effective Address
EXG Address	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	1 1 0 0	Address Register	1 0 1 0	1 Address Register
EXG Data	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	1 1 0 0	Data Register	1 0 1 0	0 0 Data Register
EXG Memory	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	1 1 0 0	Core Register	1 1 0 0	0 1 Address Register
EXT Long	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	0 1 0 0	1 0 0 0	1 1 0 0	0 0 Register
EXT Word	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	0 1 0 0	1 0 0 0	1 0 0 0	0 0 Register
JMP	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	0 1 0 0	1 1 1 0	1 1	Effective Address
JSR	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	0 1 0 0	1 1 1 0	1 0	Effective Address
LEA	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	0 1 0 0	Register	1 1 1	Effective Address

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LINK	0	1	0	0	1	1	1	0	0	1	0	1	0	Register		

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOVE Byte	0	0	0	1	Destination			Mode			Source			Register		
	Register				Mode			Mode			Register					

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOVE from SR	0	1	0	0	0	0	0	0	1	1	Effective Address					

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOVE from USP	0	1	0	0	1	1	1	0	0	1	1	0	1	Register		

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOVE Long	0	0	1	0	Destination			Mode			Source			Register		
	Register				Mode			Mode			Register					

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOVEM EA to Registers	0	1	0	0	1	1	0	0	1	Sr	Effective Address					

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOVEM Registers to EA	0	1	0	0	1	0	0	0	1	Sr	Effective Address					

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOVEP	0	0	0	0	Register			Op-Mode			0	1	Register			
Op-Mode: Word to Reg = 100, Long to Reg = 101, Word to Mem = 110, Long to Mem = 111																

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOVEQ	0	1	1	1	Register			0	Data							

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOVE to CCR	0	1	0	0	0	1	0	0	1	1	Effective Address					

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOVE to SR	0	1	0	0	0	1	1	0	1	1	Effective Address					

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOVE to USP	0	1	0	0	1	1	1	0	0	1	1	0	0	Register		

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOVE Word	0	0	1	1	Destination			Mode			Source			Register		
	Register				Mode			Mode			Register					

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MULS	1	1	0	0	Register			1	1	1	Effective Address					

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MULU	1	1	0	0	Register			0	1	1	Effective Address					





	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Shifts, Data Register	1	1	1	0	Count/Register			d	Size		i/r	Type		Register		

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Shifts, Memory	1	1	1	0	0	Type		d	1	1	Effective Address					

Shift Type Codes: AS = 00, LS = 01, ROX = 10, RD = 11

d (direction): Right = 0, Left = 1

i/r (count source): Immediate Count = 0, Register Count = 1

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
STOP	0	1	0	0	1	1	1	0	0	1	1	1	0	0	1	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SUB	1	0	0	1	Register			Op-Mode		Effective Address						

Op-Mode

B W L  
000 001 010 : Dn ← EA → Dn  
100 101 110 : EA ← Dn → EA  
— 011 111 : An ← EA → An

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SUB Immediate	0	0	0	0	0	1	0	0	Size		Effective Address					

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SUBQ	0	1	0	1	Data		1	Size		Effective Address						

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SUBX	1	0	0	1	Destination Register			1	Size		0	0	R/M	Source Register		

R/M (register/memory): register - register = 0, memory - memory = 1

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SWAP	0	1	0	0	1	0	0	0	0	1	0	0	0	Register		

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TAS	0	1	0	0	1	0	1	0	1	1	Effective Address					

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TRAP	0	1	0	0	1	1	1	0	0	1	0	0	Vector			

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TRAPV	0	1	0	0	1	1	1	0	0	1	1	1	0	1	1	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TST	0	1	0	0	1	0	1	0	Size		Effective Address					

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
UNLK	0	1	0	0	1	1	1	0	0	1	0	1	1	Register		

# 命令索引

ABCD 62,180  
 ADD 60,166  
 ADDA 61,167  
 ADDI 61,167  
 ADDQ 61,167  
 ADDX 62,169  
 AND 62,178  
 ANDI 62,179  
 ANDI to CCR 68,211  
 ANDI to SR 68,215  
 ASL 63,187  
 ASR 63,187

Bcc 65,197  
 BCGH 64,192  
 BCLR 64,192  
 BRA 65,196  
 BSET 64,192  
 BSR 65,198  
 BTST 64,192

CHK 68,209  
 CLR 62,171  
 CMP 61,172  
 CMPA 61,173  
 CMPI 61,173  
 CMPM 61,173

DBcc 65,197  
 DBRA 197  
 DIVS 61,160,177  
 DIVU 61,176

EOR 62,179  
 EORI 62,179

EORI to CCR 68,212  
 EORI to SR 68,215  
 EXG 57,59,160  
 EXT 62,171

JMP 65,195  
 JSR 65,198

LEA 57,59,159,202  
 LINK 59,204  
 LSL 63,184  
 LSR 63,184

MOVE 58,152,153,159  
 MOVE EA to CCR 68,211  
 MOVE EA to SR 68,216  
 MOVE SR to EA 68,211  
 MOVE to USP 68,217  
 MOVEA 153,155  
 MOVEM 57,59,155  
 MOVEP 57,59,157  
 MOVEQ 58,59,158  
 MULS 61,173  
 MULU 61,173

NBCD 62,182  
 NEG 61,169  
 NEGX 62,171  
 NOP 62,69,212  
 NOT 62,180

OR 62,179  
 ORI 62,179  
 ORI to CCR 68,212

ORI to SR 68,215

PEA 58,59,160,202

RESET 26,42,68,99,192,218

ROL 63,190  
 ROR 63,190  
 ROXL 63,190  
 ROXR 63,190  
 RTE 68,217  
 RTR 67,199  
 RTS 67,199

SBCD 62,181  
 Scc 65,207  
 STOP 68,92,218  
 SUB 60,168  
 SUBA 61,168  
 SUBI 61,168  
 SUBQ 61,168  
 SUBX 62,171  
 SWAP 59,160

TAS 32,62,182  
 TRAP 68,95,208  
 TRAPV 68,208  
 TST 61,172

UNLK 59,204,205

# 事 項 索 引

## あ 行

IPL (読み専用プライオリティレベル) 25  
 IPC 120  
 アーキテクチャ 6  
 ASCII 228  
 アセンブラ 138  
 アセンブラ制御命令 139,148  
 値呼び 200  
 アドレスエラー 113,218  
 アドレスエラー例外処理 108,223  
 アドレス空間 6  
 アドレス形式 7,43,144  
 アドレスストローブ (AS) 17,23,29  
 アドレススペース番号 131  
 アドレスバス 22  
 アドレス変換 130  
 アドレスレジスタ 10  
 アドレスレジスタ間接形式 46,146  
 アドレスレジスタ直接形式 45,145  
 RES (リセット) 26,41  
 RS-232C 234,236,240  
 RMS 234  
 R/W (リード/ライト制御) 17,18,23  
 アレイチェイニング 126  
  
 E (イネーブル) 18,26  
 EMS 250  
 EMS モード 244  
 EQU 148  
 イネーブル (E) 18,26  
 EPCI 120  
 EPROM ライタ 241  
 イミディエイト形式 53,148

イミディエイト命令実行時間 83  
 インデックス付アドレスレジスタ間接形式 49,146  
 インデックス付プログラムカウンタ相対形式 52,147  
 インデックスレジスタ 11,146,147  
 インブライド形式 55,144  
  
 ウェイクアップ 134  
  
 AS (アドレスストローブ) 17,23,29  
 ASE 241,245  
 ALU 8  
 ACIA 116,119  
 SSDA 116  
 SSP (スーパーバイザシステムスタックポインタ) 11,217  
 Xフラグ 77  
 H680SD 300 236  
 H680SBC 232  
 H680TR 01 235  
 Nフラグ 71  
 FIFO 164  
 FC (ファンクションコード) 28,216  
 FDOS 248  
 FDOS モード 244  
 FDC 11  
 FPCP 120  
 エミュレーション 212  
 MSB 10  
 MMU 120  
 MPCC 120  
 MPU モジュール 240  
 LIFO 164  
 LANCE 120

LSB 10  
LDS (下位データストローブ) 17, 23, 29  
演算回路 7  
演算フィールド 140

ORG 148  
オートリクエストモード 125  
オーバーフロー 74, 209  
オーバーラン 165  
オブジェクトモジュール 138, 150  
オペランド 10, 44  
オペランドサイズ 10  
オペランドフィールド 141  
オペランドフェッチ 80  
オペレーションコード 8  
オペレーションワード 43

## か行

外部クロック周波数 79  
外部割込み 101, 216  
外部割込みレベル 101  
書き込み違反 127  
拡張ファンクションユニット 8  
拡張命令 62  
格納サイクル 83  
加算命令 60, 166  
  
記号 142  
逆ポーランド記法 163  
キャリイフラグ (Cフラグ) 76  
キュー 164  
共有変数 183

タイクイミディエイト形式 55  
クリア命令 62, 171  
クロック (CLK) 28  
クロックサイクル 19  
クロックサイクル数 79

継続動作法 126  
桁移動 184  
桁移動操作命令 63  
桁送り 184  
減算命令 60, 168

コメント 139  
コメントフィールド 141  
コンディションコード操作命令 210

コンディションコードレジスタ (CCR)  
70, 152

## さ行

サイクルスチールモード 124  
サブルーチン操作命令 198  
サブルーチン呼出し 198  
算術演算命令 60, 166  
算術形桁移動命令 187  
算術シフト命令 187

CRT エディタ 250  
CRTC 116  
CLK (クロック) 28  
CCR (コンディションコードレジスタ)  
70, 152  
CCR/SR 形式 148  
システムスタック 12, 160, 162, 204  
システムスタックポインタ 10  
システム割込命令 67, 208  
システムバイト 13  
システムプログラム 220  
システムレジスタ 95  
実効アドレス 44  
実効アドレス拡張ワード 43, 48  
実効アドレス計算時間 79  
実効アドレス生成関係命令実行時間 88  
実行命令 139  
実行ユニット 7

CTC 120  
シフト機能 8  
シフト命令 63, 184  
シフト/ローテート命令実行時間 84  
Cフラグ (キャリイフラグ) 76  
ジャンプ関係命令実行時間 88  
10進定数 141, 162  
循環形桁送り命令 190  
循環バッファ 165  
上位データストローブ (UDS) 17, 23, 29  
条件セット命令 207  
条件付ブランチ命令 65, 195  
乗算命令 61, 173  
除算命令 61, 176  
シリアル通信インターフェイス 134  
シングルアドレッシング 125  
シングルオペランド命令実行時間 84  
シングルボードコンピュータ (H680 SB 01)  
232

シングルボードコンピュータシステム  
(H680 SBC) 232  
シングルステップ 41  
シングルチップ マイクロコンピュータ 8

スタック 11, 200, 202  
スタティック RAM 113  
ステータスレジスタ 13  
——の内容 215  
ステータスレジスタ操作命令 68  
スタートメント 139  
ストップ状態 92, 218  
スーパーバイザシステムスタックポインタ  
(SSP) 11  
スーパーバイザ状態 7, 11, 93  
スーパーバイザ状態フラグ 13  
スーパーバイザスタック 162  
スーパー PL/H 254  
スプリアス割込み 104

整数データ 15  
セカンダリメモリマップ 243  
セクション 150  
SECTION 151  
絶対アドレス形式 147  
絶対値記号 143  
絶対値式 143  
Zフラグ (ゼロフラグ) 73  
セマフォ 183  
セマフォレジスタ 134  
専用小形コンソール (H68 PC 01) 236

相互排除の原理 183  
相対値記号 143  
相対値式 143  
ソースプログラム 139

## た 行

ダイナミック RAM 114  
タイマ機能 134, 135  
ダイレクトメモリアクセス (DMA) 120,  
163  
多倍精度演算命令 169  
短絶対アドレス形式 50, 147

長絶対アドレス形式 51

通常状態 92

DS 149  
DMA (ダイレクトメモリアクセス) 120,  
162

DC 149  
ディジット 10  
ディスプレースメント 44, 147  
ディスプレースメント付アドレスレジスタ間  
換形式 48, 146  
ディスプレースメント付プログラムカウンタ  
相対形式 52, 147

DTACK (データ転送アノレッジ) 24,  
30

デスクリプタ 130  
デスティネーションオペランド 11  
テストアンドセット命令 182  
テスト命令 172  
データストローブ 23  
データ転送アノレッジ (DTACK) 24,  
30

データ転送命令 57, 152  
データ転送命令実行時間 80  
データバス 22, 23  
データバッキング機能 125  
データレジスタ 10  
データレジスタ直接形式 45, 145  
デバッグモジュール 240  
デュアルアドレッシング 125  
転送アドレッシング 125  
転送リクエストモード 124

同期バスインターフェイス 110  
特権違反例外処理 106, 222  
特権命令 68, 215  
トラップ発生命令 68, 208  
トラップベクタ番号 208  
トラップ例外処理 222  
トレース機能 223  
トレース処理 216  
トレースモードフラグ 12  
トレース例外処理 104, 223

## な 行

ナノ制御 9  
2重バス障害 26, 41, 101  
2進化10進数 (BCD) 14, 180  
——の真数変換 225  
2進化10進数演算命令 62, 180

2進定数 142  
2の補数表現 15  
256K バイト ダイナミックメモリモジュール  
240  
入出力ポート 135

## は 行

倍精度演算命令実行時間 90  
バイト 10  
バイト操作命令 46  
バイブライン方式 3  
バスアービトレーション 17, 25, 35  
バスエラー 25, 38, 218  
バスエラー入力信号 (BERR) 18  
バスエラー例外処理 38, 108, 223  
バスオペレーション 22, 28  
PASCAL 253  
バスグラント (HG) 17, 25, 35  
バスグラントアタノレッジ (BGACK)  
17, 25, 35  
バスサイクル 19, 36  
——の再実行 38  
バーストモード 125  
バスマスタ 37  
バス要求 (BR) 17, 25, 35  
ハードディスク装置 241  
バリッドペリフェラルアドレス (VPA)  
18, 26  
バリッドメモリアドレス (VMA) 18, 27  
バリティ 229  
番地呼び 202  
ハンドシェイク機能 135  
反復処理 197  
  
PIA 116, 119  
PI/T 120, 135  
BR (バス要求) 17, 25, 35  
BERR (バスエラー入力信号) 18  
比較命令 61, 172  
BG (バスグラント) 17, 25, 35  
BGACK (バスグラントアタノレッジ)  
17, 25, 35  
PGC 120  
BCD=2進化10進数  
ビット 10  
ビット操作機能 8  
ビット操作命令 64, 184, 192  
ビット操作命令実行時間 86

非同期バスインターフェイス 110  
非同期バス制御信号 17, 23  
非同期パラレルバス 17  
非マルチプレックスバス 17  
標準命令実行時間 82

ファンクションコード (FC) 28, 216  
VERSA 239  
VMA (バリッドメモリアドレス) 18, 26  
VPA (バリッドペリフェラルアドレス)  
18, 26  
Vフラグ (オーバーフローフラグ) 74  
FORTRAN 252  
符号拡張 49, 154, 156, 157, 159  
符号拡張機能 16  
符号拡張命令 171  
浮動小数点演算エミュレーション 213  
不当命令例外処理 106, 222  
プライマリメモリマップ 243  
ブランチ関係命令実行時間 86  
ブリデクリメントアドレスレジスタ間接形式  
47, 146  
プログラムカウンタ 13  
プログラム実行状態 93  
プログラムスタック 11, 162  
プログラム操作命令 65, 195  
プロセッサ処理状態 92  
プロセッサステータス信号線 18  
ブロック転送 161  
フロッピディスクコントロールモジュール  
240  
フロッピディスク制御ボード (H680 FD)  
234  
分画命令 195  
  
ベクタ部込み機能 7  
ベースレジスタ 151  
  
補数命令 169  
ポストインクリメントアドレスレジスタ間接  
形式 46, 145  
ホールド (HALT) 26, 38, 40  
ホールド状態 92  
ポロー 76

## ま 行

マイクロコンピュータ開発支援装置 238  
マイクロコンピュータシステム 232

マイクロプログラム制御方式 7  
 マイクロアセンブラ 252  
 マルチタスティング 241  
 マルチチップマイクロコンピュータ 3  
 マルチプロセッサ 219  
 マルチプロセッサシステム 182  
 マルチブロック転送 126  
 マルチレジスタ関係命令実行時間 88

未実装命令 69, 106, 212  
 未実装命令例外処理 106, 222  
 未定義セグメント 127

無条件ジャンプ命令 65, 195  
 無条件ブランチ命令 65, 195  
 無条件分岐命令 195

命令サイクル 19  
 命令シーケンス 20  
 命令実行時間 78  
 命令制御部 7  
 命令プロダ 9  
 命令トラップ例外処理 106  
 命令プリフェッチ機能 19  
 命令レジスタ 8  
 メモリマップド I/O 7, 18, 110  
 メモリマネージメント機能 18

文字定数 142  
 MOS-LSI 2  
 モニタボード (H 680 MN 01) 234

## や 行

優先割込み構造 18  
 ユーザシステム スタック 11, 162  
 ユーザシステム スタック ポインタ (USP) 11  
 ユーザ状態 7, 11, 93  
 ユーザバイト 13  
 USING 151  
 UDS (上位データストローブ) 17, 23, 29

予約語 142

## ら 行

ライトサイクル 30  
 ラベルフィールド 139

RAM ボード (H 680 DM 12) 234  
 リアルタイムデバッグ装置 241  
 リメントラント 204  
 リカーシブ 204  
 リセット (RES) 26, 41  
 リセット例外処理 99, 220  
 リードサイクル 29  
 リードモディファイライトサイクル 32, 39  
 リードライト制御 (R/W) 17, 18, 23  
 リフレッシュ動作 114  
 リロケータブル 150  
 リロケータブルオブジェクトモジュール 251  
 リンクチェイニング 126  
 リンケージエディタ 138, 150, 251

例外状態 92  
 例外処理 12, 95, 217  
 例外処理時間 91  
 例外ベクタ 97  
 例外ベクタアドレス 95  
 例外ベクタ番号 95  
 レジスタ 6  
 レジスタ構成 9  
 レジスタ返還 154  
 レジスタリスト 155

68010 216  
 ローゲート命令 63, 184, 190  
 ロードモジュール 138, 150, 251  
 ROM 113  
 ロングワード 10  
 論理演算命令 62, 178  
 論理形桁移動命令 184  
 論理シフト命令 184  
 論理ベースアドレス 131

## わ 行

ワード 10  
 割込みアタノレッジサイクル 103, 115  
 割込みベクタ発生回路 115  
 割込みベクタ番号 26  
 割込みマスク 13  
 割込みマスク情報 101  
 割込み優先レベル 25  
 割込み例外処理 101, 221  
 ワンボードコンピュータシステム 235



著者の現職

喜田祐三 株式会社 日立製作所  
萩原吉宗 株式会社 日立製作所  
岩崎一彦 株式会社 日立製作所

マイクロコンピュータシリーズ 14

68000 マイクロコンピュータ

定価 3,400 円

昭和 58 年 3 月 30 日 発行

©1983

著 作 者

喜  
田  
祐  
三

萩  
原  
吉  
宗

岩  
崎  
一  
彦

喜  
田  
祐  
三

発 行 者 海 老 原 熊 雄

発 行 所 丸 善 株 式 会 社

郵便番号 103 東京都中央区日本橋二丁目3番10号

印刷 中央印刷株式会社・製本 株式会社 松谷社

3355—2755—7924

森 亮一 監修

マイクロコンピュータシリーズ 〈A5〉

---

1		マイクロコンピュータ	
2	P. R. Rony 著 森監訳, 田島 他訳	マイクロコンピュータ教科書Ⅰ	定価 2,200 円
3	〃	マイクロコンピュータ教科書Ⅱ	定価 3,300 円
4	〃	マイクロコンピュータ教科書Ⅲ	定価 3,600 円
5		マイクロコンピュータ教科書Ⅳ	続 刊
6		マイクロコンピュータ教科書Ⅴ	
7	森 亮一 監修	汎用 マイクロプロセッサ	定価 3,200 円
8	森 亮一 監修	ワンチップマイクロコンピュータ	定価 3,200 円
9	森 亮一 監修	ビットスライスマイクロプロセッサ	定価 3,200 円
10	国 分 明 男 著	マイクロコンピュータメモリ	続 刊
11	寺 田 浩 詔 監訳	Z-80 マイクロコンピュータ	定価 3,300 円
12	禿, 喜 田 著 田 辺, 藤 岡	16ビットマイクロプロセッサ	定価 3,300 円
13	D. I. Porat 他著 柏 本 浩 光 訳	デ ィ ジ タ ル 技 術 入 門	定価 4,000 円
14	喜田, 萩原, 岩崎 著	68000 マイクロコンピュータ	定価 3,400 円
15	田 辺 皓 正 編著	8086 マイクロコンピュータ	4 月 刊
16	寺 田, 禿 前 著 宮 本, 前 田	Z 8000 マイクロコンピュータ	近 刊

---

定価は変更することがありますのでご了承下さい

昭和 58 年 2 月現在





定價 3,400 円

3355-2755-7924